

Dipartimento di Informatica, Bioingegneria,  
Robotica ed Ingegneria dei Sistemi

---

**On the Design and Implementation of  
Next Generation Cyber Ranges**

by

Enrico Russo

Theses Series

**DIBRIS-TH-2021-XX**

---

DIBRIS, Università di Genova

Via Opera Pia, 13 16145 Genova, Italy

<http://www.dibris.unige.it/>

**Università degli Studi di Genova**

**Dipartimento di Informatica, Bioingegneria,**

**Robotica ed Ingegneria dei Sistemi**

**Ph.D. Thesis in Computer Science and Systems Engineering**

**Computer Science Curriculum**

**On the Design and Implementation of  
Next Generation Cyber Ranges**

by

Enrico Russo

January, 2021

**Dottorato di Ricerca in Informatica ed Ingegneria dei Sistemi**  
**Indirizzo Informatica**  
**Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi**  
**Università degli Studi di Genova**

DIBRIS, Univ. di Genova  
Via Opera Pia, 13  
I-16145 Genova, Italy  
<http://www.dibris.unige.it/>

**Ph.D. Thesis in Computer Science and Systems Engineering**  
**Computer Science Curriculum**  
(S.S.D. ING-INF/05, INF/01)

Submitted by Enrico Russo  
DIBRIS, Univ. di Genova

Date of submission: October 2020

Title: On the Design and Implementation of Next Generation Cyber Ranges

Advisor: Alessandro Armando<sup>1</sup>, Gabriele Costa<sup>2</sup>

<sup>1</sup>Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi, Università di Genova, Genova (Italy)

<sup>2</sup>SysMA Unit, IMT School for Advanced Studies, Lucca (Italy)

Ext. Reviewers: Paolo Prinetto<sup>†</sup>, Silvio Ranise<sup>\*</sup>, Luca Viganò<sup>‡</sup>

<sup>†</sup>Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino (Italy)

<sup>\*</sup>Security & Trust Research Unit, FBK-Irst, Trento (Italy)

<sup>‡</sup>Department of Informatics, King's College London, London (UK)

## Abstract

*Cybersecurity attacks are on the rise, and a competent workforce able to face real-life threats is urgently needed. Their training requires practical learning opportunities and, in particular, hands-on exercises. Cyber Defense Exercises (CDX) can meet the demand for realistic, hands-on training. Unfortunately, running a CDX requires dedicated infrastructures, namely Cyber Ranges, to host the training scenarios. Furthermore, building the computing infrastructure is only the first step. Indeed, the design, verification, and deployment of scenarios are costly and error-prone activities. The reason is that a misconfiguration in the scenario can compromise the exercise and the training goals. The result is that CDX of real-world complexity are so expensive that only a limited number of organizations can afford them.*

*In this thesis, we consider the problem of designing an effective and usable Cyber Range capable of hosting training scenarios for the next generation of security experts. We start our investigation by reconsidering common training activities such as Capture the Flag (CTF) competitions. In particular, we present our experience with a non-formal training activity for university students that we organized. The goal was to test the overall effectiveness of acquired skills and analyze the challenge development process.*

*By leveraging this experience, we focus on the implementation of a Cyber Range. We present CRACK, a framework for the (i) design, (ii) model-based verification, (iii) generation, and (iv) automated testing of cyber scenarios. At the core of our approach stands the Scenario Definition Language (SDL) that extends TOSCA, an OASIS standard for the specification and orchestration of virtual cloud infrastructures. Our SDL allows for the defining and formally verifying specification of the scenario elements and their interplay. Verified scenarios are automatically deployed and tested to check if they are ready to be played.*

*Finally, we use our Cyber Range to create a scenario replicating a realistic system involving the use of the emerging Fog computing paradigm. As a side effect of this activity, we introduce DIOXIN, an extension of the considered Fog operating system that mitigates found weaknesses.*

# Contents

<b>Chapter 1</b>	<b>Introduction</b>	<b>5</b>
1.1	Context and Motivations . . . . .	5
1.2	Contributions . . . . .	7
1.2.1	Training Security Skills: the ZenHackAcademy Experience . . . . .	8
1.2.2	Building Next Generation Cyber Ranges . . . . .	9
1.2.3	Cyber Ranges on the Field . . . . .	10
1.3	Structure of the Thesis . . . . .	11
<b>Chapter 2</b>	<b>Training Security Skills: the ZenHackAcademy Experience</b>	<b>13</b>
2.1	Capture The Flag competitions . . . . .	13
2.2	The ZenHackAdemy . . . . .	15
2.2.1	Autumn 2017, first pilot . . . . .	15
2.2.2	Autumn 2018, second edition . . . . .	16
2.3	Results . . . . .	18
<b>Chapter 3</b>	<b>Building Next Generation Cyber Ranges with CRACK</b>	<b>21</b>
3.1	Case Study: ACME Corp . . . . .	23
3.2	Preliminaries . . . . .	24
3.2.1	Cyber ranges and training . . . . .	24
3.2.2	Infrastructure provisioning . . . . .	26
3.2.3	Datalog . . . . .	29

3.3	Scenario Definition Language . . . . .	30
3.3.1	TOSCA integration . . . . .	30
3.3.2	SDL types . . . . .	33
3.3.3	Behavior and runtime . . . . .	38
3.3.4	Access pattern language . . . . .	38
3.4	Introducing CRACK . . . . .	39
3.4.1	Overview of the approach . . . . .	40
3.4.2	Scenario Verification . . . . .	41
3.4.3	Deployment and testing . . . . .	44
3.5	CRACK Demo . . . . .	48
3.5.1	Design . . . . .	49
3.5.2	Verification . . . . .	50
3.5.3	Deployment and testing . . . . .	52
3.5.4	Execution . . . . .	54
3.5.5	Evaluation . . . . .	54
<b>Chapter 4</b>	<b>Cyber Ranges on the Field</b>	<b>59</b>
4.1	Practical Fog Computing . . . . .	59
4.1.1	Fog Computing . . . . .	60
4.1.2	Fog Applications . . . . .	61
4.1.3	Cisco IOx . . . . .	62
4.1.4	Cisco NBI . . . . .	63
4.2	Motivating Scenario . . . . .	64
4.2.1	A smart agriculture system . . . . .	64
4.2.2	Fog implementation . . . . .	64
4.3	Security Model . . . . .	69
4.3.1	NBI Security Features . . . . .	69
4.3.2	Weaknesses of NBI . . . . .	70

4.4	Methodology . . . . .	71
4.4.1	Overview of the approach . . . . .	71
4.4.2	Implementation . . . . .	76
4.4.3	Experimental Evaluation . . . . .	79
<b>Chapter 5</b>	<b>Related Work</b>	<b>82</b>
5.1	Hands-on and CTFs . . . . .	82
5.2	Cyber Ranges . . . . .	83
5.3	Fog Security . . . . .	86
<b>Chapter 6</b>	<b>Conclusions</b>	<b>88</b>
	<b>Bibliography</b>	<b>90</b>

# Chapter 1

## Introduction

### 1.1 Context and Motivations

Each year, cybersecurity reports filed by governments [Sis19, UK 19], private companies [Mic20, Che20a, Fir20], and non-profit organizations [ISA20, CLU20] review cyber incidents and provide key insights about the cyber threat landscape. Most of these reports agree that we are facing a significant increment in the number and complexity of threats and make us suggest that no organization, big or small, is immune to a disruptive cyber attack. Moreover, the ability to prevent or defend against such attacks is strongly linked to cybersecurity professionals' readiness and skills. These conditions result in an ever-growing demand for well trained security experts.

Practical learning opportunities are essential for cybersecurity operators to achieve an adequate cyber risk awareness level and the skills to counter threats. In this sense, the European Network and Information Security Agency (ENISA) recommends including hands-on exercises as a good practice in designing and implementing the cybersecurity strategy of nations [ENI16b].

Nowadays, the two most widely recognized hands-on activities in the field of cybersecurity training are *(i)* Capture the Flag (CTF) competitions and *(ii)* Cyber Defense Exercises (CDX), also known as live-fire cyber exercises.

CTF competitions come in different shapes, e.g., Jeopardy [DEC<sup>+</sup>11] and Attack-and-Defense (A/D) [TDG<sup>+</sup>17a]. Despite the format, CTFs feature a game-like approach and involve individuals or small teams solving *challenges*. Each challenge focuses on a specific topic, such as Binary Exploitation, Reverse Engineering, Web Exploitation, Cryptography, or Forensics. Typically, challenges are made up of an artifact (e.g., an executable file, a network packets capture or a filesystem image) or a single virtual machine running a vulnerable system. Solving these challenges represents an opportunity to learn hacking techniques (and necessary countermeasures), strengthen problem-solving skills, and gain valuable hands-on practice. Some of the reasons why



CTFs are becoming so popular are (i) the variety of topics covered, (ii) the gradual progression in the exercise difficulty, (iii) the standardized ruleset, and (iv) the few resources required to host and run them. On the other hand, the main issue with CTFs is that they focus on a single topic per time. Therefore, they are unable to cover the complexity of security assessments and the comprehensive process of cyber defense.

CDXs [MLM<sup>+</sup>07, SO18] are exercises in which teams of participants (blue teams) manage and defend a compound of realistic Information Technology (IT) and Operation Technology (OT) infrastructures. These exercises enable to train operators to detect and mitigate large scale cyber-attacks executed by specific teams (red teams) simulating hackers' activities. The main difference from CTFs lies in the complexity of the operational environment, namely the training *scenario*, and the educational goals. In particular, CDXs emphasize realistic training scenarios and focus on improving technical and soft skills, e.g., communication, teamwork, and decision-making under stress, in the cybersecurity domain. For these reasons, they also allow organizations to identify gaps and areas for development in their processes and technologies. Hosting and running such exercises require specialized infrastructures, namely *Cyber Ranges*, providing the necessary training environment.

Cyber Ranges can host highly realistic cyber exercises: large scale ICT infrastructures consisting of hundreds of interconnected devices, each supporting the protocol stack, operating system, and applications of choice can be emulated on commercial hardware or by leveraging virtualization technologies. Cyber Ranges are being used to carry out large scale cyber exercises routinely.

A prominent example is the Locked Shields cyber exercise [CCD19], which NATO conducts on a yearly basis since 2010. In Locked Shields, blue teams must defend a given (emulated) ICT infrastructure from attacks performed by a red team. The emulated ICT infrastructures used in Locked Shields consist of around four thousand virtualized systems and are executed by means of a dedicated Cyber Range.

Currently, Cyber Ranges and CDXs are possibly the best hands-on training experience. Nevertheless, these exercises are extremely resource-intensive and require a great deal of planning before execution [Dew18]. As discussed in [VVO<sup>+</sup>17], implementing such a cyber exercise is the result of a sequence of complex operations. Nowadays, these operations are mostly carried out manually by security experts. The authors of [VVO<sup>+</sup>17] identify several factors behind the high effort, time, and costs for creating a cyber exercise. Among them, the trial-and-error approach is a major one.

The required implementation process is schematically depicted in Figure 1.1. The possible inputs are some *initial requirements*, usually consisting of a high-level, conceptual design or a storyboard, being the result of a preliminary brainstorming. These elements drive the *theater design* whose objective is the definition of the infrastructure hosting the cyber exercise. This phase yields a blueprint of the theatre. The *scenario design* refines the blueprint by adding non-infrastructure elements, e.g., vulnerabilities and (teams of) participants. Then, the *objectives*

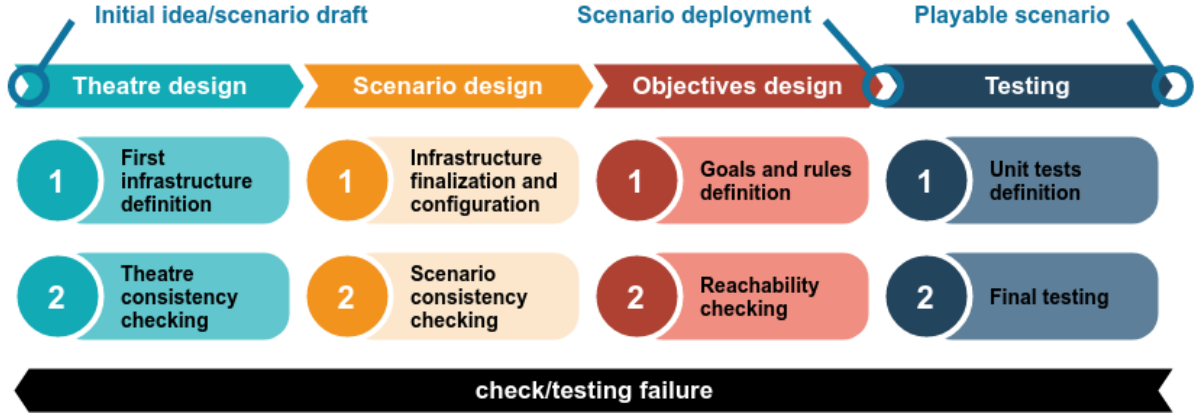


Figure 1.1: The scenario implementation process.

*design* introduces the goals and rules of the cyber exercise. All these phases require a validation step aiming at the detection and elimination of inconsistencies. Clearly, a failed validation makes the design process backtrack to an earlier phase. Eventually, the validated blueprint of the scenario reaches the *scenario deployment* phase. This operation translates the blueprint into a set of directives which are fed to an *Infrastructure-as-a-Service* (IaaS) [MG11] provider. The result is a running infrastructure implementing the blueprint. Unfortunately, there is no guarantee that the actual system behaves as expected, i.e., it preserves the properties validated on the blueprint. The main reasons are the lack of a formal semantics and the abstraction level of the model. The first issue arises because an element of the blueprint is instantiated with a virtual machine running a real OS and real software. The second issue resides in the IaaS deployment approach that requires making explicit some values that might be absent at the level of abstraction of the blueprint, e.g., network addresses. For these reasons, the deployed scenario undergoes a testing phase to check that the cyber exercises can be properly executed. When also the test validation is passed, the scenario is ready for the execution.

## 1.2 Contributions

This thesis takes its cue from the importance of hands-on exercises to fill the shortage of cyber security skills. First, we consider CTF competitions and their acknowledged effectiveness in providing valuable hands-on experiences also with limited infrastructural and organizational resources. These features make them the ideal training solution for organizations of all types and sizes, including academic ones. Then, we review Cyber Ranges as they are currently the most comprehensive and versatile systems to support organizations in training operators, identifying and addressing gaps in the cybersecurity domain. In particular, we focus our attention on the

hosted scenario which represents one of the key elements to maximize the outcome of such systems. Finally, we evaluate the effectiveness of Cyber Ranges not only as a training environment, but also as resources for hosting security testbeds. To this aim, we use a Cyber Range to design and deploy a realistic scenario and carry out a security evaluation against a connected, physical device.

To assess our contribution, we detail the issues and the related research questions for each covered topic.

### 1.2.1 Training Security Skills: the ZenHackAcademy Experience

As previously discussed, cybersecurity attacks are on the rise, and the need for a competent workforce is increasing. However, this huge demand is currently unsatisfied. This issue is often referred to as the *CyberSecurity Skills Shortage* (CSSS), which ENISA analyzes in an annual report that focuses on the status of cybersecurity education in the European Union [ENI16a]. Briefly, they explain that this shortage is due to a quantitative issue, i.e., the insufficient supply of professionals to meet the requirements of the job market, but also to a *qualitative* one, i.e., the inadequacy of professional skills to meet the market's needs. In particular, they argue that *“many of the current issues in cybersecurity education could be ameliorated by redesigning educational and training pathways that define knowledge and skills that students should possess upon graduation and after entering the labour market”*.

The not-for-profit foundation Global Cyber Security Center summarizes in [Zan19] a year-long exploratory research on CSSS. Like [ENI16a], they highlight that, between the correlates of the shortage, the unqualified candidates is a prominent one. They also detail that the main factor behind the CSSS is due to the education and training system and report that *“simply put, most educational institutions do not prepare students for a career in cybersecurity”*.

For these reasons, the academic institutions must have a significant commitment in reducing the CSSS. To this aim, they must design hands-on training activities in scenarios that are as close as possible to the real-world ones. CTF challenges are a great opportunity to achieve this goal, since they encourage students to think as an attacker does, thus creating more awareness on the modalities and consequences of an attack. Unfortunately, changing the topics taught in a university curriculum is not always an easy task, since curricula are often the result of much different balances and mediation among faculty members. However, triggered by the valuable outcomes of CTFs, in 2017 we introduced in our department some hands-on activities, outside the official classes, and offered *non-formal* meetings on ethical hacking to expose students to new trends and directions.

We distinguish between *formal* and *non-formal* according to the following definitions [Tob92]. Formal learning is official, structured, organized by public or private organizations and ends with an official certification (e.g., university credits). On the other hand, non-formal education is any

structured and organized learning which does not necessarily lead to recognized qualifications or identified diplomas.

We report on a non-formal educational activity, discussing student recruitment, training organization, and results we achieved. In order to evaluate these aspects, we defined the following research questions.

RQ1) Did this non-formal training help students in improving their competencies?

RQ2) Did this non-formal training increased students' interest in cybersecurity?

### 1.2.2 Building Next Generation Cyber Ranges

Since the need of advanced cybersecurity training is now perceived by a wide audience (e.g., the civil sector), a number of Cyber Range solutions, both commercial (e.g., Cyber Range in a Box [Bel14], Cyberbit Range [Cyb19], and NetWars [SAN19]) and open source (e.g., ADLES [CdLGHK18], CyRIS [BPT<sup>+</sup>18], and KYPO [VOC<sup>+</sup>17]), have recently been put forward.

Most of these proposals rely on virtualization to support scalability and reconfigurability. Yet, live-fire cyber exercises running against emulated ICT infrastructures of real-world complexity can be afforded only by a limited number of organizations. As observed by [VVO<sup>+</sup>17], this is mostly due to the fact that the design, development, and deployment of scenarios of real-world complexity are error-prone, time-consuming activities that require the involvement of highly specialized personnel. Such an effort is always necessary, since playing the same scenario more than once drastically reduces its training effectiveness. Thus, although a large infrastructure helps, the overall quality of a cyber exercise mostly depends on its good design.

To illustrate the level of complexity, consider the difficulty of installing the necessary software and configuring each and every client, server, router, gateway in a scenario comprising hundreds of such devices. In theory, the resulting infrastructure, namely the *theater*, can be developed by means of existing infrastructure development frameworks (e.g., see Terraform [Has20], Ansible [Red20], Chef [Che20b], and Puppet [Pup20]). However, the theater is just the first step as it must then be turned into an appropriate training scenario. This step is fraught of difficulties as it requires, e.g., the injection of vulnerabilities (e.g., misconfigurations and software bugs), the definition of the teams' goals, and the testing of their actual reachability. For instance, the injection of too many (or too easy to discover) vulnerabilities may lead to a scenario that can be trivially solved, whereas the injection of too few (or too difficult to discover) vulnerabilities may lead to a frustrating experience for the participants. Unfortunately, a mistake made during the design may be discovered only at a later stage, e.g., during the development, the deployment or even the execution of the scenario.

The issues discussed above are summarized by the following research questions:

- RQ3) Can (and to what extent) main-stream development methods for virtual infrastructures be extended to support the development of Cyber Range scenarios?
- RQ4) Can (and to what extent) state-of-the-art modeling and automated verification techniques be used to drive the scenario development process?
- RQ5) Can designers reuse (parts of) previously developed scenarios without compromising the overall quality of the training activity?

### 1.2.3 Cyber Ranges on the Field

The effectiveness of cyber defenders to contain attacks has as the main cornerstone their hands-on training. Nevertheless, the overall defense of an organization against cyberattacks, also known as the *cybersecurity posture*, is an even more challenging process. In addition to the training activity, the cybersecurity posture encompasses the deployed security solutions, the security policies in place, and the collective security status of all the installed software, hardware, network and services.

Evaluating the cybersecurity posture of an organization become of paramount importance to identify the attack surface along with the effectiveness and efficiency of the defensive mechanisms. Notice that, a proper assessment relies also on the invasive testing of systems in production, such as executing a Vulnerability Assessment and Penetration Test (VAPT). Moreover, the above tests must be performed as a regular basis and every time the system under evaluation is updated with a new configuration or with a new component.

Unfortunately, there are many situations where invasive test on live systems is not advisable, even when it is possible. For example, think about industrial cyber-physical systems, i.e., systems where the operational technology is integrated with traditional IT infrastructures. A fully VAPT includes potentially disruptive actions, e.g., exploiting vulnerabilities or performing denial-of-service attacks, and executing it against these systems can certainly highlight the security gaps but also create unwanted interruptions to the manufacturing activity. For these reasons, many times the cybersecurity posture assessment is incomplete and limited in recurrence so resulting inaccurate and ineffective.

Considering the features offered by Cyber Ranges and the issues presented above we propose the following research question:

- RQ6) Are Cyber Ranges flexible enough to support the security assessment of critical assets by replicating their production environment?

.

## 1.3 Structure of the Thesis

The thesis is structured as follows.

### **Chapter 2 - Training Security Skills: the ZenHackAcademy Experience**

In this chapter, we introduce CTFs and detail our experience in using them as hands-on during an academic educational activity on the cybersecurity topic. First, we review the main features of CTFs and the different formats of their challenges. Then, we describe the training approach we have used and how we have integrated it with the traditional academic classes. Finally, we present the results that have been summarized from the survey submitted to the students who participated in this activity.

### **Chapter 3 - Building Next Generation Cyber Ranges**

In this chapter, we introduce Cyber Ranges with a particular focus on their training scenario. We start by describing a case study and the working example. Then, we recall some preliminary notions, and we present our Scenario Definition Language. The above notions and SDL are required for introducing CRACK, our framework for cyber scenarios. Finally, we demonstrate and evaluate the framework.

### **Chapter 4 - Cyber Ranges on the Field**

In this chapter, we examine Cyber Ranges as facilities to host testbeds for performing security assessments. In particular, we focus on testing Cisco IOx, a mainstream operating system used in Fog Computing deployments. First, we introduce some background on Fog Computing and Cisco IOx. Then, we extend the case study introduced in the previous chapter adding components for executing a smart irrigation system. This scenario allowed us to test the security model of Cisco IOx and highlight the main weaknesses. To mitigate the found issues, we present DIOXIN, our proposal to improve the security of Cisco IOx through runtime monitoring. Finally, we validate the effectiveness of our proposal using the same testbed hosted by the Cyber Range.

### **Chapter 5 - Related Work**

In this chapter, we present related work. First, we compare the ZenHackademy experience with other activities involving hands-on exercises. Then, we discuss solutions comparable to our CRACK framework. Finally, we compare DIOXIN with other proposals introducing policies and runtime monitoring in IoT and Fog deployments.

### **Chapter 6 - Conclusions**

The material in this manuscript is based on the following papers.

- Enrico Russo, Gabriele Costa, and Alessandro Armando. Scenario Design and Validation for Next Generation Cyber Ranges. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–4. IEEE, 2018
- Luca Demetrio, Giovanni Lagorio, Marina Ribaud, Enrico Russo, and Andrea Valenza. ZenHackAdemy: Ethical Hacking@DIBRIS. In *CSEDU (1)*, pages 405–413, 2019
- Enrico Russo, Gabriele Costa, and Alessandro Armando. Building Next Generation Cyber Ranges with CRACK. *Computers & Security*, 95:101837, 2020
- Enrico Russo, Luca Verderame, Alessandro Armando, and Alessio Merlo. DIOXIN: Runtime Security Policy Enforcement of Fog Applications. *International Journal of Grid and Utility Computing*, 2020. In press

## Chapter 2

# Training Security Skills: the ZenHackAcademy Experience

In this chapter we detail the different formats and rules of CTFs (Section 2.1). Then we present how we introduced these competitions and related exercises in our academic institution (Section 2.2). Finally, we discuss the lesson learned from this experience (Section 2.3).

### 2.1 Capture The Flag competitions

Below, we list the main formats for CTF competitions.

**Jeopardy.** This competitions [CBB14, FSHB17] involve multiple categories of challenges, each of which contains a vulnerability. Participants, often grouped into teams, must exploit these vulnerabilities to find hidden *flags*, i.e., strings in a given format. The knowledge of a flag proves that the corresponding challenge has been successfully solved. Participants do not directly attack each other. They enroll into an online platform to access the challenges and submit the found flags to gain points. Jeopardy CTFs have a fixed duration, which is usually from one to a few consecutive days. Competitions in this format allow students to think *adversariarly* [HHM<sup>+</sup>17], i.e., to think as an attacker would, and this form of gamification motivates them to learn by doing.

**Attack/Defense (A/D).** In A/D CTFs, teams run an identical machine, or a small network, injected with vulnerable services. In this format, the goal of each team is to find and exploit the vulnerabilities in opponent's machines, while fixing or mitigating flaws in their own. Unlike jeopardy CTFs, flags associated to vulnerable services are dynamic. The scoring system, namely



the *scoring bot*, updates them regularly and teams lose points if their services are not up when the scoring bot contacts them. On the other hand, compromising a service of other machines enables a team to steal the corresponding hidden flags and gain points. All in all, the score is computed considering the services availability together with the number of stolen flags. A/D CTFs last a few hours and usually teams can take advantage of extra time before the start of the competition to analyze services and patch found flaws. Although A/D CTFs are more demanding to play, they allow participants to gain experience with both offensive and defensive related skills.

**King of the Hill (KotH).** KotH CTFs [BHL18] differs from A/D format as the teams do not run their own machine. Instead of individual machines, a KotH CTF relies on a central set of servers. Teams need to identify vulnerability of the above server and find out how to exploit them. After taking the control of a system, a team must be able to prevent other participants to hacking it and scoring points.

**Build-it/Break-it/Fix-it (BIBIFI).** BIBIFI CTFs [RHP<sup>+</sup>16, WCC18] ask build-it teams to write software, which is subsequently attacked by break-it teams. BIBIFI competition consist of three phases. The first one, namely *build-it*, requires small development teams to build software according to a given specification that includes security goals. The second phase, namely *break-it*, requires teams to find defects in other teams' submissions. Reported defects benefit the break-it team's score and penalize the build-it team's score. The final phase, namely *fix-it*, asks builders to fix bugs and thereby get some points back.

In all formats of CTFs, a dynamic scoreboard shows the progress of the contest, listing the teams and their scores. At the end of the competition, the scoreboard is frozen, and the top three teams are listed as winners.

Then, a phase dedicated to the publication of *write-ups* follows. Who resolved a given challenge can write a short post and detail the steps of the adopted solution. From a training perspective, this task is of primary importance. On the one hand, participants can improve reporting skills arranging and summarizing their solutions. On the other hand, it allows to compare different techniques applied to solve the same challenge.

The reference website for the CTF competitions is CTFtime [CTF20]. In particular, it constantly updates a list of the past, current, and future events. For each CTF competition, a difficulty score is assigned. The difficulty score is computed by evaluating some parameters such as players' feedback and the scores of the teams. All the registered teams participate in a global ranking. After every CTF, a team receives new points corresponding to their score in the CTF normalized according to the overall difficulty score.

Table 2.1 shows CTFtime statistics for the years from 2015 to 2019. In particular, we report the number of CTFs, the registered teams, and registered academic teams, i.e., teams affiliated with

Table 2.1: CTFtime statistics.

	2015	2016	2017	2018	2019
CTFs	79	107	141	153	198
Teams	7271	10590	14983	18237	24555
Academic Teams	327	499	659	826	1045

academic institutions.

The numbers of Table 2.1 show that the interest about CTF competitions has been constantly growing over the last years. Moreover, they highlight that more and more universities create their own teams.

## 2.2 The ZenHackAdemy

ZenHack is a CTF team we founded in 2017 and made up of students and researchers interested in cybersecurity. The team is based in Genova<sup>1</sup>, Italy, and is hosted by the Department of Informatics, Bioengineering, Robotics, and Systems Engineering (DIBRIS) of the University of Genova.

Besides participating in competitions as an academic team, our activity in ZenHack aims to foster practical skills in cybersecurity. In particular, having experienced the educational potential of CTFs, we considered proposing challenges as hands-on training to the undergraduate students.

As anticipated in Section 1.2.1, modifying the content of official curricula is a difficult and time-consuming process. For this reason, in October 2017, we decided to assess the interests in CTF topics and the effectiveness of their exercises by starting some non-formal training, outside official lectures. To manage all these activities, organized with the help and expertise of ZenHack, we created the ZenHackAdemy.

In Section 2.2.1, we present how we organized the first pilot of the ZenHackAdemy. In Section 2.2.2, we detail the second edition and refinements based on the previous experience.

### 2.2.1 Autumn 2017, first pilot

We scheduled a calendar of weekly meetings reserving a slot without official courses. Lessons was organized to cover different topics, ranging from web security to binary analysis, from net-

---

<sup>1</sup>The dialect name of Genoa is *Zena*

work analysis to cryptography. Each session included a brief theoretical introduction on the topic of the lesson and a discussion on how to solve some of the related challenges.

Participation was somewhat encouraging with around 50 participants during the first meeting. As expected, this number decreased over time when the complexity of the covered topics increased.

At the end of the training path, we organized an on-site Jeopardy CTF event. We choose CTFd [Chu20] as the platform for hosting the local CTF. CTFd is an open-source software designed to support CTF organizers handling exercises publication, participants enrollment, and flags submission.

Thirty-two students attended the first on-site CTF that lasted for few hours and exposed them to solve challenges on the topics covered during the training path. The winner was a 2nd-year bachelor student in Computer Engineering with no prior experience in computer security.

Finally, as a noteworthy result, some of the participating students joined us in the ZenHack team.

## 2.2.2 Autumn 2018, second edition

At the beginning of the academic year 2018/19, we re-proposed the activities of the ZenHackAdemy trying to strengthen the experience of the previous year, as described below.

**Students enrollment.** In the Autumn 2018, once again, we scheduled a 10-week non-formal training supported by the ZenHackAdemy. In this academic year, we were able to reach a large number of students w.r.t. first pilot. In particular, the holder of the Computer Security course recognized the importance of such hands-on activities and decided to add this non-formal training as a companion to the official track.

**Lectures organization.** In this second edition, we published all the exercises on a CTFd instance dedicated to the training activity. Thus, students could practice with the platform of the final CTF from the beginning of the learning path. At the same time, we were able to track their involvement and progress. In detail, analyzing the CTFd data, we have determined that 126 students registered to the platform, and 90 were effectively active and tried to submit some flags.

Activities started on October 2018, according to the schedule shown in Table 2.2.

In the first lesson, we introduced the importance of *ethics* and *legislation* when practicing ethical hacking. Then, we presented some basic Linux commands and tools that can be useful for solving challenges. We closed the lesson proposing exercises for practicing the *lateral thinking* [DBZ70], a necessary mindset for dealing with CTF exercises.

The second meeting focused on network security and forensics. In particular, we presented

Table 2.2: Calendar of the activities, Autumn 2018.

12/10	Ethical hacking and Linux basics
19/10	Network security and forensics
26/10	Web security (client-side)
09/11	Basics on machine learning
16/11	Basics on cryptography
23/11	Exercises
30/11	Web security (server-side)
07/12	Binary analysis
14/12	Binary analysis (cnt.)
20/12	Final on-site CTF

Wireshark [the20i], a mainstream network protocol analyzer. The challenges to solve consisted of finding flags on network traffic captures.

We scheduled two meetings on web security to present both client-side vulnerabilities, e.g., Cross-Site Scripting (XSS), and server-side ones, e.g., SQL Injection (SQLi). For each of the above class of vulnerabilities, students performed attacks on the Google Gruyere [Goo20] platform.

In this edition, we also decided to cover some basics of machine learning, particularly an introduction to *adversarial* machine learning [HJN<sup>+</sup>11]. We provided the students with challenges proving how to fool simple classifiers.

We dedicated a lesson to cryptography by introducing the underlying principles. The proposed challenges consisted of breaking simple encryption schemes that are improperly implemented.

Finally, we devoted the last two meetings to a brief introduction to the binary reversing of Linux executables. The first meeting covered executable “life-cycle”, i.e., how binaries are linked and loaded, and some basics of disassembly process. In the second meeting, we covered the cornerstone of reverse engineering binaries. This was mainly carried out by leveraging some tools for dynamic analysis, e.g., *strace*, *ltrace* and *gdb*.

**On-site CTF.** On December 20<sup>th</sup> 2018, we organized the on-site CTF having four prizes for first places. Out of the 90 students active on the training platform, 71 took part in the local CTF, 22 of which enrolled in the Computer Security course.

We prepared 21 exercises in all categories introduced during the training. Challenges were of two different types. Seven of them were specific for the students who have attended the Computer Security course. Their score was static, i.e., 10 points each. All remaining exercises used dynamic

scores, i.e., the score started from 500 points and was decremented with each new submitted solution.

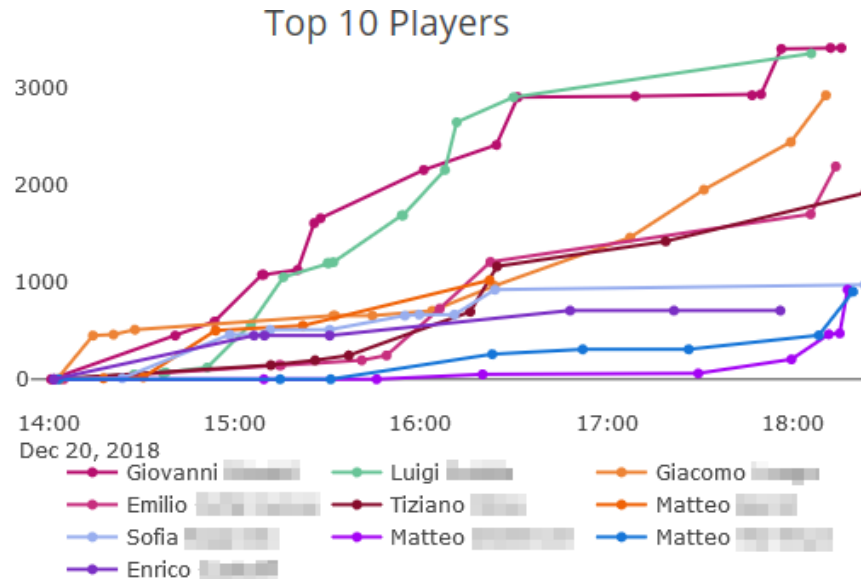


Figure 2.1: Final scoreboard.

Figure 2.1 shows the final scoreboard, with the top 10 participants and their progress during the CTF. In particular, only 1 out of the top 4 winners belong to Computer Security students and nobody else in the top 10 participants.

## 2.3 Results

We administered a short anonymous survey consisting of multiple-choice questions, 5-point scale questions, and a final open-ended question for any feedback.

We received 36 responses that correspond to the 40% of our sample, considering the 90 active students on the training platform. Below we detail the most relevant questions and the related answers.

A question (*SQ1*) of the survey asks students why they attended ZenHackAdemy activities and proposes two different answers: 1) *mandatory*, for Computer Security students, and 2) *interested in the topic*, for all the students. Respondents could select both answers, and 13 (36%) of them declared to be Computer Security students, but 31 out of 36 (86%) selected the second option

A second question (*SQ2*) concerns the background of respondents, 20 (55%) did not have any prior experience in the field.

Two questions (*SQ3* and *SQ4*) concerned the participation in the CTFs. In particular, 31 out of the 36 (86%) respondents participated in the on-site CTF, and 16 (44%) declare they will participate in other CTFs in the future, 9 (25%) would like, but they have no time, and only 2 declare they will not.

We formulated two questions (*SQ5* and *SQ6*) that asked students to self-assess their competence on the topics of the lessons (see Section 2.2.2) *before* and *after* the ZenHackAcademy activities. Figure 2.2 shows on the top how participants self-evaluated themselves on different topics, based on their prior knowledge, and, on the bottom, their self-evaluation on the same topics after the training.

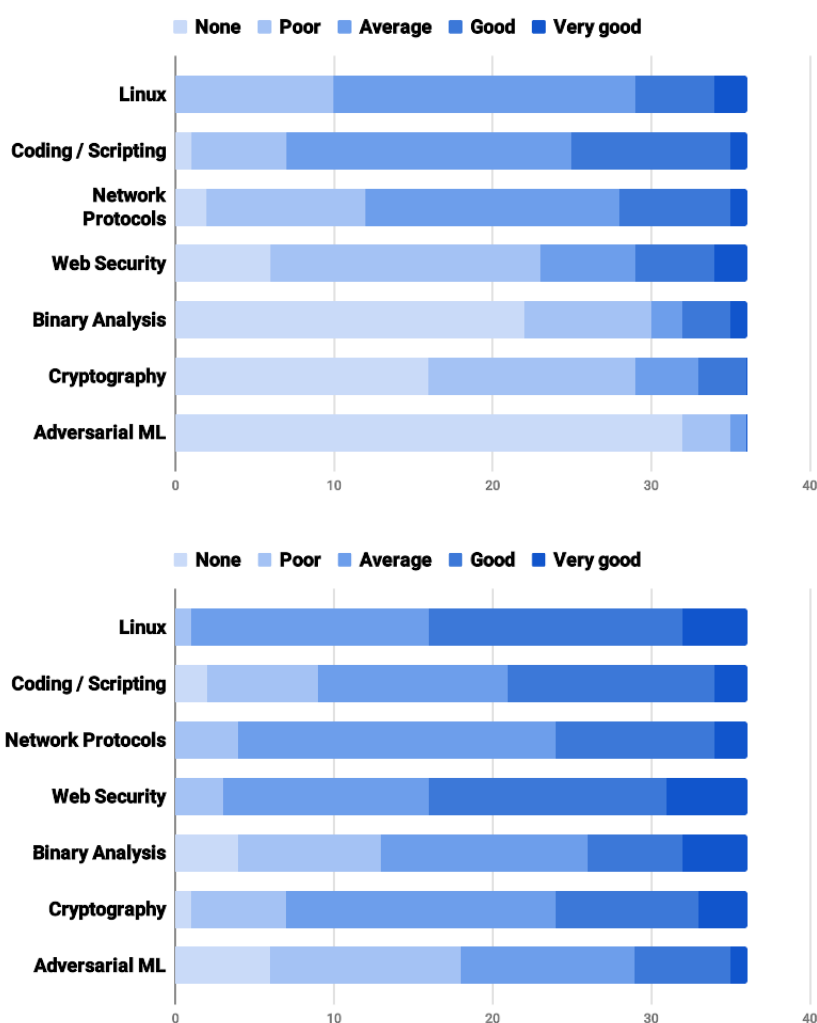


Figure 2.2: Students self-evaluations before the training (top) and after the training (bottom).

Moreover, we asked (SQ7) “How did ZenHackAdemy activities influence your opinion on: (a) Computer Security, (b) Ethical Hacking, (c) CTF, (d) ZenHackAdemy meetings?”. Figure 2.3 shows the answers.

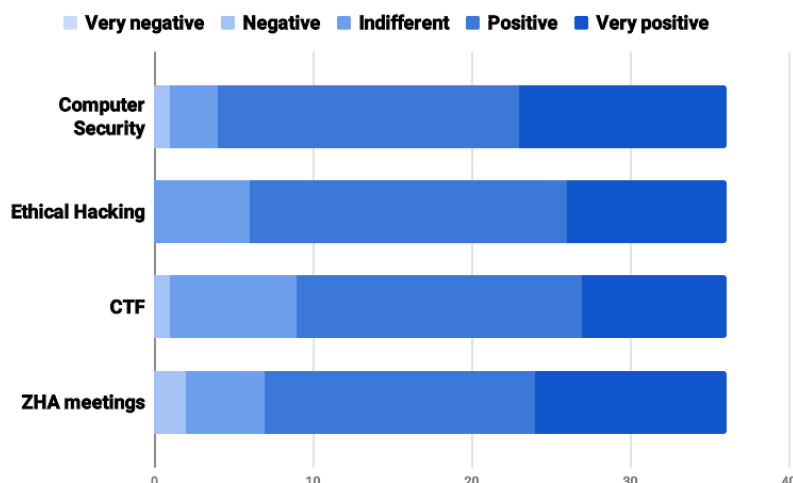


Figure 2.3: Students’ opinions on *Computer Security*, *Ethical Hacking*, *CTF*, *ZenHackAdemy meetings*.

**Answer to RQ1** By comparing the results of SQ5 and SQ6 summarized in Figure 2.2, we can notice an overall improvement concerning the skills of the participants. In particular, we observe that there is a shift towards an *average* or *good* level of self-evaluation. Moreover, only a small number of participants declare to know nothing (*none*) on the topics proposed in the list after our lessons. We can indeed claim that the non-formal meetings of the ZenHackAdemy allowed students to improve their skills: some learned new concepts, others improved their prior understanding.

**Answer to RQ2** Results of the questions SQ1 and SQ2 show that this non-formal training is attractive regardless of whether they are mandatory with official courses and having a priori knowledge on the cybersecurity subjects. Answers to the questions SQ1 and SQ2 reported that lessons had motivated a significant number of students to participate in CTFs. The results summarized in Figure 2.3 highlight an overall positive impression on the proposed topics. All in all, these findings reveal that this non-formal training activity can significantly increase students’ interest in cybersecurity.

## Chapter 3

# Building Next Generation Cyber Ranges with CRACK

Cyber Ranges are the state-of-the-art systems for hands-on cybersecurity training. Their success is mainly due to their capability to support scenarios of high complexity and realism. However, as detailed in Section 1.2.2, creating such scenarios is a demanding process and requires highly specialized personnel. For this reason, Cyber Ranges can only be used by a limited number of organizations and often without exploiting the full capabilities of these systems.

In this chapter we present the *Cyber Range Automated Construction Kit (CRACK)*. CRACK supports the (i) design, (ii) automated verification and (iii) automated testing of complex Cyber Range scenarios of real-world complexity. CRACK covers all the phases of the scenario development process as we detail below.

**Design.** We define *CRACK SDL*, a Scenario Definition Language based on TOSCA [OAS19], an infrastructure specification language standardized by OASIS. As we will see, specifications expressed in CRACK SDL can be readily composed and reused and this greatly simplifies the design process. Moreover, since CRACK SDL is an extension of TOSCA, its integration with the existing infrastructure design technologies comes with no additional effort.

**Verification.** The CRACK SDL type system allows for the automatic verification of the scenarios against several design errors, e.g., incorrect hardware/software bindings. More importantly, we show how a CRACK SDL specification can be translated into a corresponding Datalog specification which can be automatically checked by off-the-shelf Datalog engines. This allows the user to verify the CRACK SDL specifications against the training objectives.

**Deployment.** We show how a CRACK SDL specification is automatically translated into a sequence of instructions for a virtualization environment. Executing these instructions leads to the fully automated instantiation of all the elements of the scenario. As we will see, this process



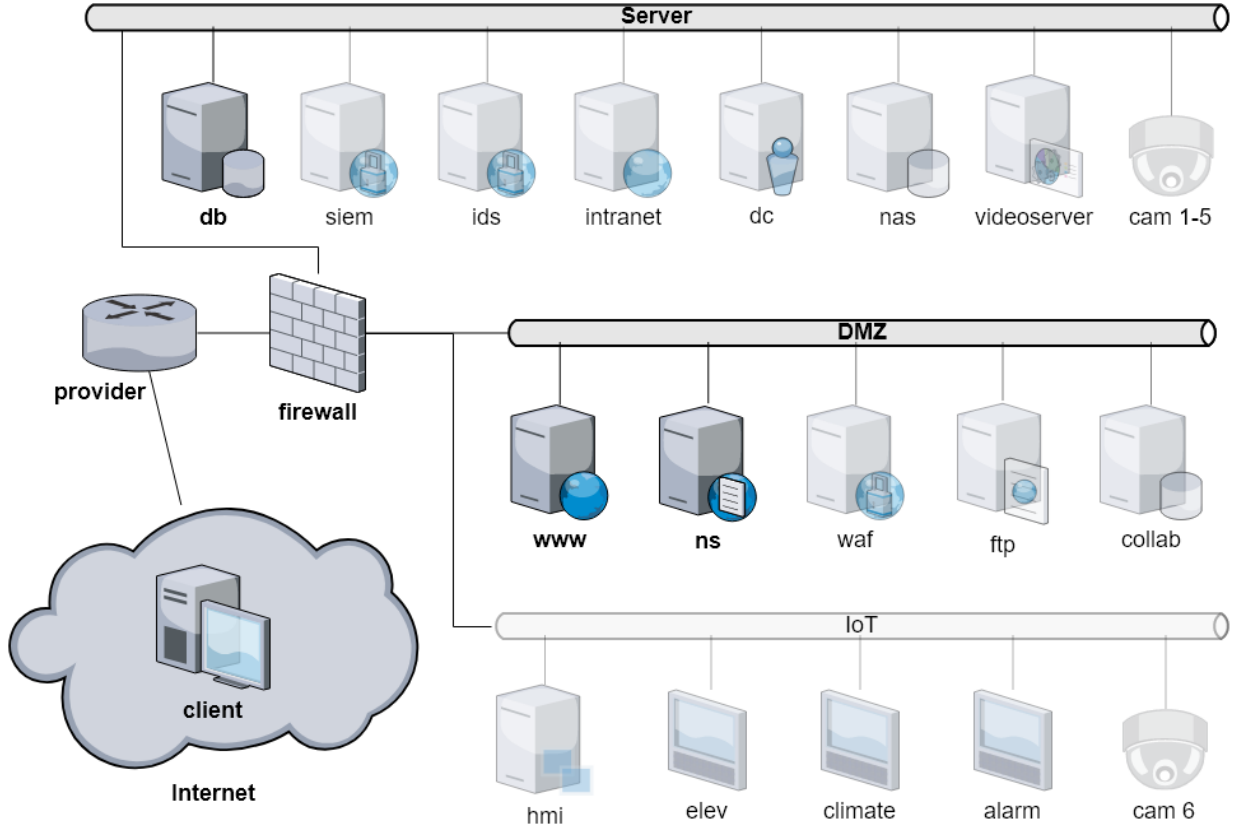


Figure 3.1: Case study infrastructure.

can take place on any TOSCA-compatible infrastructure virtualization platform.

**Testing.** We show that verification traces generated by the Datalog engine can be automatically turned into test cases for the scenario. The execution of the test cases checks whether the properties verified on the SDL specification (cf. Verification) are also enjoyed by the scenario at runtime.

As a further contribution, we introduce a case study involving four scenarios based on the same infrastructure and sharing the same goal (i.e., data exfiltration). Yet, the scenarios are affected by distinct vulnerabilities that allow for different training objectives. The case study and, in particular, Scenario 1 will be used as a working example throughout the chapter in order to illustrate the functionalities of CRACK. Also, we use the experimental results to answer the research questions of Section 1.2.2 and to confirm the effectiveness of our approach.

### 3.1 Case Study: ACME Corp

In this section, we introduce ACME Corp, a case study based on a fictional ICT infrastructure depicted in Figure 3.1. It consists of a segmented network where each segment hosts services and devices related to a specific task: (i) *Server* contains the internal services (i.e., not meant to be publicly accessible), (ii) *DMZ* contains the public services (i.e., exposed to the outside world) and (iii) *IoT* connects field devices (i.e., sensors, actuators and controllers). These three networks lay behind a *firewall* protecting the perimeter of the company. The firewall is intentionally left open toward the DMZ to allow remote connections. A domain name server (DNS), called *ns*, translates the symbolic names of the DMZ hosts into their actual IP addresses. The infrastructure is connected to the public Internet through the backbone of the Internet service provider.

The infrastructure of Figure 3.1 is the stage for the execution of four scenarios. All of them involve a *blue team* and a *red team*. The blue team has the generic goal to protect the ACME Corp assets (e.g., data and services). The red team has the specific goal to exfiltrate data from the private database residing on *db*. Playing the role of the ACME Corp IT security department, the blue team has full access to the internal network, whereas the red team has only access to the public Internet through a remote *client* machine.

We briefly discuss the four scenarios below.

**Scenario 1 (Host Security 1)** In this scenario the red team can achieve its goal by exploiting some security weaknesses in the configuration of the *www* server, including:

1. *www* hosts a Content Management System (CMS) that uses the administrator credentials [The20b] for authenticating to the Database Management System (DBMS) running on *db*.
2. *www* runs an HTTP server exposing the home directories of the users; this enables a dictionary-based enumeration of the existing accounts [The20d].
3. One of the enumerable users of *www* has a weak password, i.e., a password that is subject to brute-force attack [The20e].
4. The administrator of *www* is exposed to an *Escalation of Privileges* (EoP) vulnerability [The20g]. An EoP vulnerability allows the attacker to acquire the privileges of the administrator.

**Scenario 2 (Web Security)** In this case, the red team can exploit the weaknesses given below.

1. Same as in Scenario 1.

2. The remote debugging interface of the CMS is active, so allowing for Python commands injection [Pal20].
3. The application server runs with unnecessary administrator privileges [The20b], so exposing the database access credentials.

**Scenario 3 (Host Security 2)** In this scenario, the red team can exploit the following weaknesses and vulnerabilities of the *www* server.

1. Same as in Scenario 1.
2. *www* runs a version of the PHP interpreter suffering from a Remote Code Execution (RCE) vulnerability [The19d] that, e.g., allows the attacker to create a backdoor on *www*.

**Scenario 4 (Network Security)** In the last scenario, the red team can achieve the goal exploiting the following misconfigurations and vulnerabilities.

1. *www* runs an Apache server with an active HTTP reverse proxy module [The20a]. The module is misconfigured to be an open proxy [The20c], i.e., it forwards HTTP requests from the Internet to the internal servers, e.g., connected to the *Server* network.
2. *db* runs a version of the Webmin [Cam00] software suffering from a RCE vulnerability [The19e].

In the rest of the chapter we only focus on some of the elements discussed above. In particular, we only consider the part of the infrastructure highlighted in Figure 3.1. We will use it and Scenario 1 as the working example through the chapter. In Section 3.5.4, we will discuss the effort needed to design the other scenarios by reusing (parts of) Scenario 1.

## 3.2 Preliminaries

In this section, we briefly recall the notions that are relevant for correctly understanding the content of the chapter.

### 3.2.1 Cyber ranges and training

According to the National Institute of Standards and Technologies [NIS18] Cyber Ranges are “interactive, [...] representations of an organization’s local network, system, tools, and applications that are connected to a simulated Internet level environment.” The goal of a Cyber Range

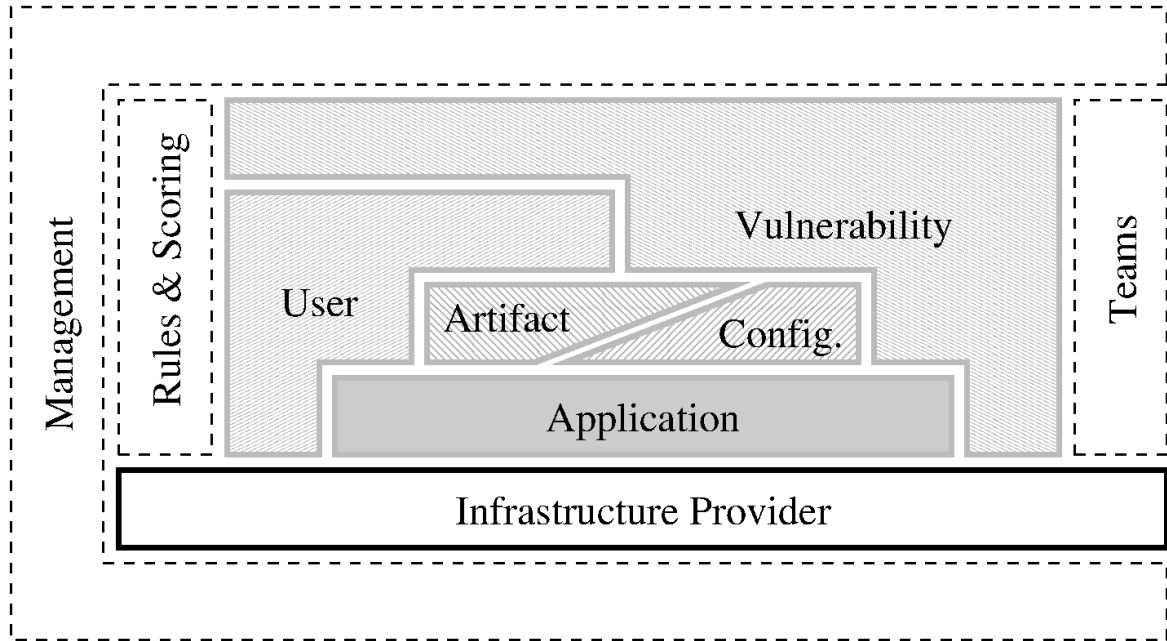


Figure 3.2: Cyber Range logical scheme.

is to “provide a safe, legal environment to gain hands-on cyber skills and a secure environment for product development and security posture testing.”

In the spirit of the above definition, in this work, we propose a logical structure of a generic Cyber Range. Such a structure is depicted in Figure 3.2. The *management* facilities support the planning and execution of the activities conducted within the Cyber Range. This may imply the monitoring of the Cyber Range activities possibly driving their evolution along a given story line.

The hands-on training is carried out within an *infrastructure*, consisting of a pool of (virtualized) networks, computers, and applications hosted by some provider. We call such an infrastructure the *theater*. Intuitively, we consider part of the theater all the elements that are not specific to the training objectives, i.e., any object that is passively or marginally involved in the current exercise. For instance, the routing infrastructure of a layered network architecture, being only responsible for the exchange of messages, is a part of the theater.

On top of the theater, the *scenario* is the collection of all the items that are relevant for the hands-on activity. It is worth noticing that a neat distinction between theater and scenario does not exist in general (e.g., in terms of the involved technologies). In fact, the very same object can be part of either the theater or the scenario depending on its role in the exercise. Nevertheless, the distinction is useful to better characterize the role of the elements appearing in a cyber exercise.

Although it is not a general rule, a typical scenario involves some *applications*, e.g., a remote shell or a CMS. Such applications are customized with *configurations*, e.g., the remote authenti-

cation method, and *artifacts*, e.g., an authentication key, that plays a role in the scenario. Similarly, *user* accounts can be included. A user can be related to all of the elements mentioned above, e.g., the administrator of a service (application) has also access to its files (artifacts and configurations). Finally, *vulnerabilities* must be injected to enable the attackers' exploits. Vulnerabilities are a cornerstone of every scenario and they can involve any of the elements discussed above, e.g., a user setting a weak password or an application failing in sanitizing an input.

Beside the scenario and the theater, the gameplay facilities, i.e., the *scoring* and *rule* systems as well as the *team* support, must be implemented. For instance, in a scenario some servers cannot be attacked (engagement rules) or the blue team loses points when a certain service becomes unavailable (service level agreement). Similarly, the teams may be required to operate through some terminals that only provide a limited number of security tools. These elements cannot be developed once for all as they are scenario-dependent. Nevertheless, they can be occasionally reused, e.g., the same scoring system might apply to a certain category of exercises. Although they are not properly part of the theater, these elements must be also included in the Cyber Range.

Since in this work we deal with the design and verification of the scenarios, we only focus on the part of the elements of Figure 3.2. In particular, we will reason about the scenario elements (light gray boxes). Moreover, we will discuss the infrastructure provider technologies that support the deployment of theaters and scenarios. Instead, we will skip the presentation of the management and gameplay elements (dashed boxes).

### 3.2.2 Infrastructure provisioning

In this section, we briefly recall the two infrastructure provisioning paradigms involved in our proposal.

**Infrastructure-as-a-Service (IaaS)** IaaS [MG11] aims at providing a flexible and reconfigurable infrastructure development platform. In particular, an IaaS provider allows for a direct control over machines, operating systems, applications, and networking. By relying on virtualization technologies, IaaS platforms hide the underlying, physical infrastructure (a.k.a. bare-metal).

In a Cyber Range, each theater consists of many different elements including hosts (e.g., servers and desktop clients), software (e.g., operating systems and applications) and network facilities (e.g., routers and firewalls). Conveniently, IaaS providers expose APIs for creating, deleting, and reconfiguring these elements. This makes IaaS a suitable paradigm for defining and deploying a Cyber Range theater. In this setting, the building blocks of any theater are virtual machines (for computing and storage) and virtual switches, routers, networks, and network ports (for implementing the network infrastructure). Although some elements may not allow virtualization, a virtual network can also be connected with some physical resources outside the IaaS platform.

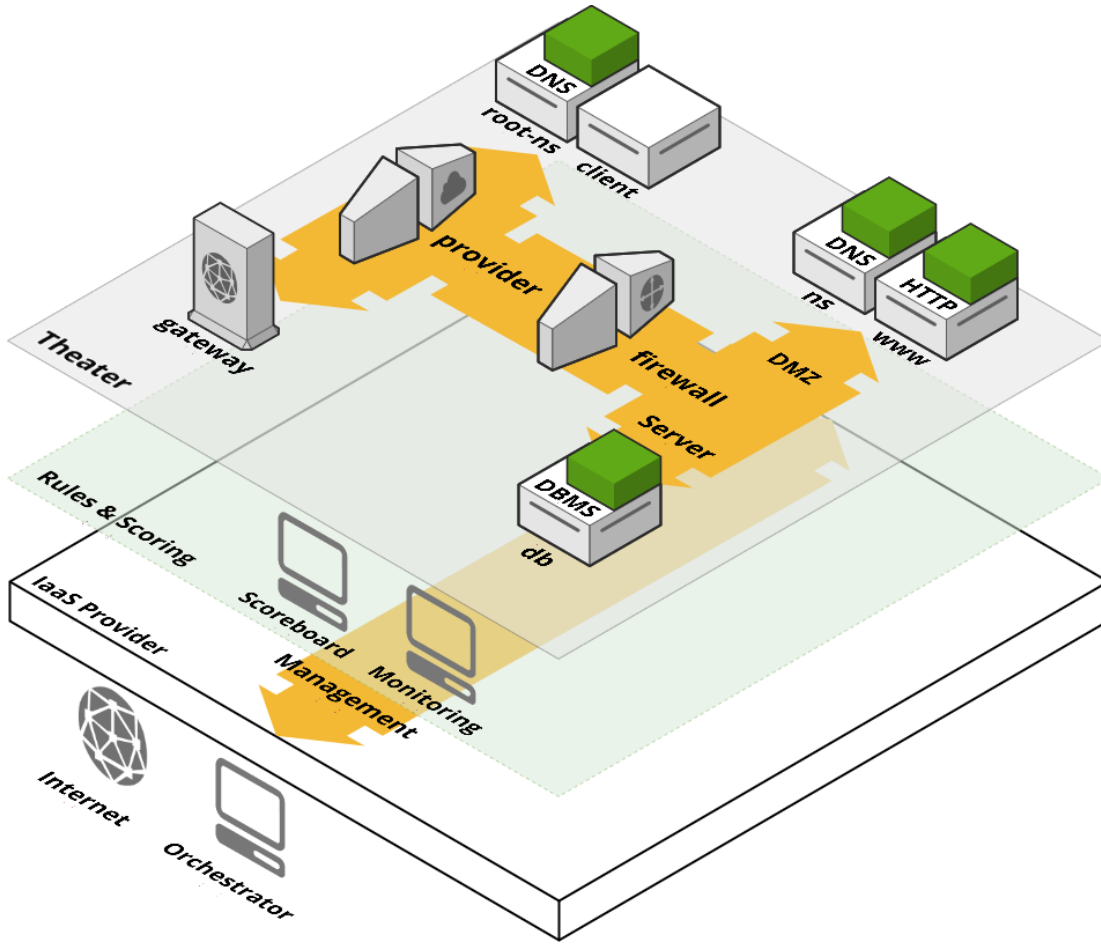


Figure 3.3: Layered view of the case study theater deployed over an IaaS provider.

For instance, an infrastructure can be connected to the Internet through a gateway.

Figure 3.3 represents the deployment of the theater of our case study (see Section 3.1) on an IaaS provider. Briefly, the deployment proceeds bottom-up and each layer contributes to the instantiation of the above ones. We use transparency to denote that items residing at a higher layer are accessible to the layers below, e.g., the monitoring system can access the theater elements. Instead, lower layers are accessible only if they expose some functionalities, e.g., the management network is open to monitoring and scoreboard.

All the elements of the theater are virtual with the only exception of the Internet which is only partially simulated. The real Internet is accessible through a gateway that directly connects to the external network. Moreover, the IaaS supports for the virtualization of part of the Cyber Range facilities, such as the scoreboard and rule monitoring services. These facilities stay on a different layer as they are not accessible from within the theater. Instead, they operate through

a management network that is responsible for the cross-layer connectivity. Such connectivity is necessary to *orchestrate* the theater, i.e., to create and configure the virtual infrastructure.

**Infrastructure-as-Code (IaC)** On an IaaS provider, instantiating the infrastructure as in Section 3.1 requires the following operations.

1. Create the virtual networks, e.g., Server and DMZ.
2. Create all the virtual machines, e.g., db and www.
3. Connect each virtual machine to the proper networks, e.g., db to Server.
4. Install all the operating systems and applications, e.g., DBMS on db.
5. Finalize the infrastructure by adding configurations, artifacts, and users.

All these operations are carried out by submitting the corresponding commands, e.g., via some APIs, to the IaaS provider. Nevertheless, as the complexity of the infrastructure increases, handling these design and deployment operations without a systematic approach quickly becomes cumbersome and error-prone.

In the last years *Infrastructure-as-Code* (IaC) [ABDN<sup>+</sup>17] emerged as the main infrastructure design approach. A IaC framework uses a specification language to model the desired infrastructure. A provisioning tool, called *orchestrator*, takes as input the specification and automatically deploys the infrastructure on an IaaS provider. We propose the following example to clarify the structure of a generic IaC specification language.

Figure 3.4 provides a class diagram representation of the infrastructure appearing in the working example. The box at the bottom, labeled with *Primitive*, contains (some of) the primitive classes defined by a generic IaC provider. These classes abstractly define the building blocks of the infrastructure, e.g., machines and networks.

The *Network* class allows for the creation of virtual networks. Each virtual network is a collection of virtual subnetworks, i.e., the *Subnetwork* class. A virtual subnetwork is labeled with two properties, i.e., *address* and *netmask*, that specify the network address and netmask of the subnetwork.

The *Compute* class represents a generic host, e.g., a virtual machine. An instance of *Compute* must declare its *image*, i.e., the installed OS, *flavor*, i.e., the hardware profile, and *init\_script*, i.e., the instructions to correctly configure the host. There can also be dependencies between *Compute* objects. For instance, the www server depends on the db server. Typically, the orchestrator is responsible for resolving the existing dependencies, e.g., by creating the *Compute* objects in the right order.

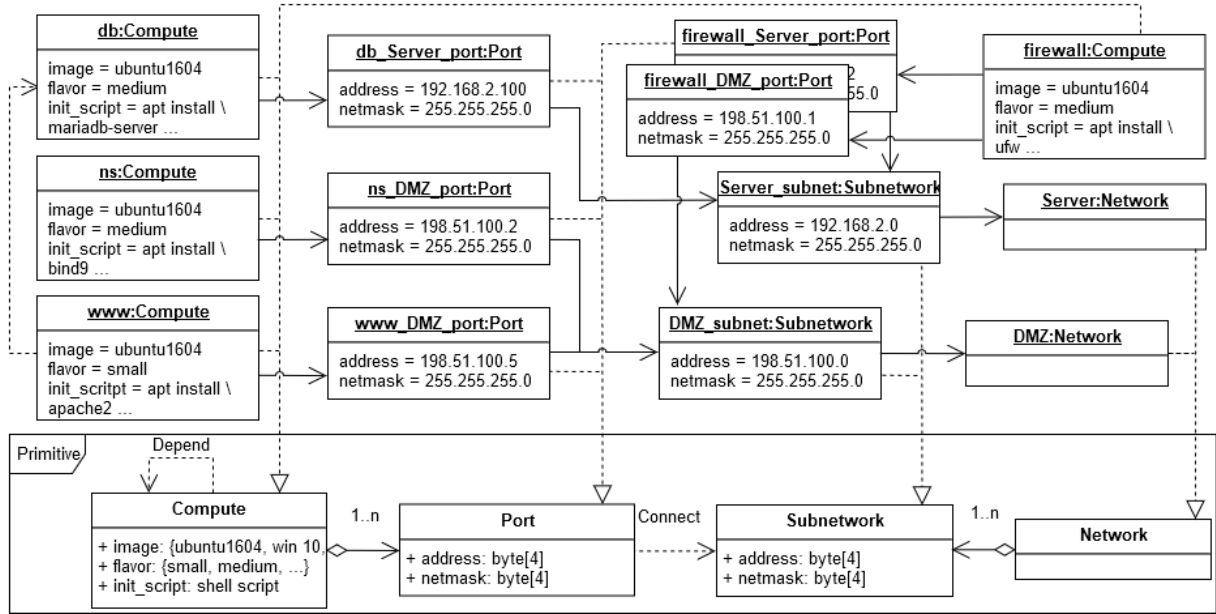


Figure 3.4: A generic IaC specification for the working example theater.

Finally, Compute objects can be connected to one or more subnetworks. This behavior is modeled by the *Port* class that defines a generic network port for connecting to a subnetwork. Each port can also carry a (fixed) IP address specified in the *address* and *netmask* properties. In the diagram, the *db\_Server\_port* and *www\_DMZ\_port* are instances of the *Port* class and connect the *db* and *www* server to the two subnetworks with addresses 192.168.2.100 and 198.51.100.5, respectively.

### 3.2.3 Datalog

Datalog [CGT90] is a declarative logic programming language which enjoys efficient algorithms for query resolution. A Datalog specification consists of a list of *facts* and *clauses*. A fact  $P(a_1, \dots, a_n)$  states that a predicate  $P$  is satisfied by the elements of a tuple  $(a_1, \dots, a_n)$ . A clause  $T : -T_1, \dots, T_n$  states that a term  $T$  can be inferred from the terms  $T_1, \dots, T_n$ , called the *premises* of the clause. Terms are also predicates, but, unlike facts, they can contain *variables*, e.g.,  $A, B, X, \dots$

A Datalog query  $T?$  is evaluated by an engine, i.e., a solver, against a specification to decide whether  $T$  is entailed by the given facts and clauses in the specification. When this is the case, the Datalog engine returns the list of facts and clauses, namely the *proof trace*, that have been applied to validate the given query.



$P(a, b).$	$1: P(b, c).$
$P(b, c).$	$2: P(a, b).$
$Q(A, B) :- P(A, C), P(C, B).$	$3: Q(a, c) :- P(a, b), P(b, c).$
$Q(X, c)?$	$Q(X = a, c)$

Figure 3.5: A Datalog specification with a query (left) and a proof trace (right).

To exemplify, consider the Datalog specification on the left of Figure 3.5. The specification consists of two facts and one clause. Moreover, we append a query at the end of the specification. The query is valid if an assignment to  $X$  can be found that satisfies  $Q(X, c)$ . This is trivially true for the query and a possible proof trace is given on the right of Figure 3.5. The proof ends by finding an assignment, i.e.,  $X = a$ , that satisfies the query. To obtain this result, the engine applied the (only) clause in the specification (line 3) by instantiating its variables. The right-hand side of the clause contains the premises that are available in the initial part of the trace (lines 1 and 2).

The decision problem for a Datalog query is P-complete in the size of the Datalog specification [DEGV01].<sup>1</sup> This ensures that the verification process scales well even with large specifications.

### 3.3 Scenario Definition Language

The design phase aims at generating a suitable blueprint of the scenario, covering all the relevant aspects from the infrastructure description to the objectives of the cyber exercise. Although the infrastructure has an important role, there are other aspects to consider when designing a scenario. In general, IaC is not meant to model these components and must be extended to support them. Our *Scenario Definition Language* (SDL) builds on TOSCA [OAS19], a prominent IaC language, but it introduces several new elements that we describe in this section. Briefly, we carry out three extensions, i.e., (i) we define new, scenario-specific node and relationship types, (ii) we introduce two special properties to support the verification and testing process and (iii) we implement a novel query language based on *access patterns*.

#### 3.3.1 TOSCA integration

The *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [OAS19] is a YAML-based [BKEI08] OASIS standard language for designing the topology and the life-cycle of a cloud application. A TOSCA-enabled IaaS provider must have a suitable TOSCA orches-

<sup>1</sup>The study of the Datalog fragments that admit efficient solvers is an active research field, e.g., see [GGV02].

trator.<sup>2</sup> TOSCA implements the concepts of Figure 3.4 by means of a rich type system. This will be further discussed in Section 3.4.2. Briefly, the main constituents of TOSCA are the following.

*Node types.* They define an infrastructure component, e.g., a server or a network, or a component element, e.g., a software installed on a server. A node type can include *properties*, *attributes*, *capabilities* and *requirements*. Properties represent some static, node-specific feature, e.g., the hostname. Attributes resemble properties, but they are used to store a value that is set by the orchestrator after the instantiation, e.g., think of a dynamically assigned IP address. Requirements and capabilities define what the node needs and (optionally) provides to the others. Requirements and capabilities mainly serve as the joints for the relationships (see below).

*Relationship and capability types.* They are used to connect nodes and, as it happens for node types, can include properties and attributes, e.g., the credentials for the authenticated service exposed by the node we are connecting. A relationship has a direction, and it connects the requirement of a source node to the capability of a target node. Moreover, each requirement can put a constraint on the types of both the target node and capability. For instance,<sup>3</sup> a WordPress web application requires to connect to (i) a database (ii) endpoint, i.e., a network database. To model this, the WordPress node includes a requirement *database\_endpoint*. The *database\_endpoint* requirement constrains the type of the target node to be *Database* and the type of the target capability to be *Endpoint*. These two constraints capture (i) and (ii), respectively.

*Interfaces.* Nodes and relationships may have *interfaces*. An interface defines a custom operation to be invoked by the orchestrator. Two kinds of interfaces exist, i.e., *standard* and *on demand*. A standard interface defines a task related to the life-cycle phases of a node (e.g., create, start, and stop). For instance, one can add a standard, *create* interface to a compute node to ask the orchestrator for installing a certain software package when the node is created. Instead, on-demand interfaces introduce new tasks. The orchestrator permits to invoke the tasks through the definition of a new workflow. For instance, the on-demand interface can be used to implement an application-specific logic (see [RBL19, § 7.3.2]).

A *node template* is a specification of a cloud application obtained through the composition of the elements mentioned above. In particular, each element is obtained by instantiating its base, namely *normative*, type. Roughly speaking, the TOSCA normative types provide a set of primitive classes (see Figure 3.4). Designers can define their own types by extending the normative types. As discussed in [BS14], the type inheritance enables some well-know mechanisms, e.g., type substitution and reuse, that simplify the design process.

**Example 1.** Consider the diagram depicted in Figure 3.6. It is an excerpt of the specification for the working example introduced in Section 3.1. In particular, it specifies the infrastructure

---

<sup>2</sup>Existing TOSCA-enabled orchestrators often accept a slightly extended version of the TOSCA standard, i.e., a TOSCA dialect. If not differently stated, the examples in this work refer to the ARIA TOSCA dialect [ARI19].

<sup>3</sup>See [OAS19] for more details.

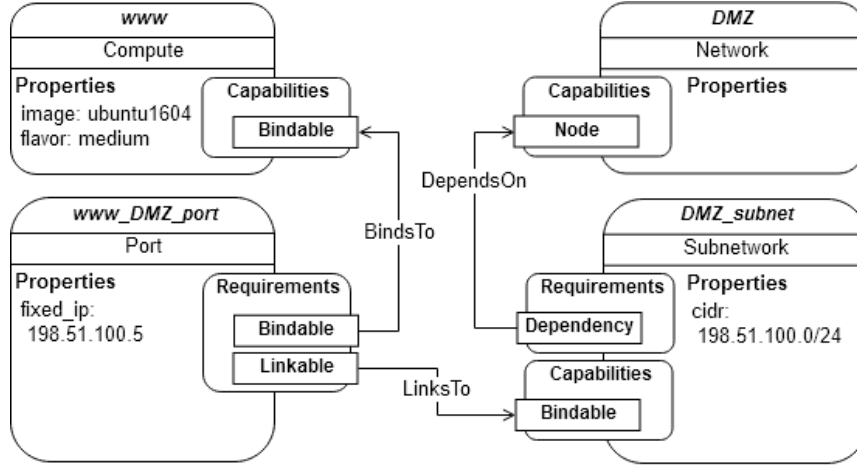


Figure 3.6: An excerpt from the (TOSCA-style) diagram for the working example.

of the web server (*www*) and the hosting network (*DMZ*). The *www* server runs on a virtual machine, an instance of the *Compute* node type. The hardware configuration of *www* and its operating system image are set by using the *flavor* and *image* properties, respectively. The *DMZ* network is an instance of the *Network* node type. A *DMZ* subnetwork, instance of *Subnetwork* node type, is in relationship, *DependsOn*, with the *DMZ* network and allows to specify its block of IP addresses in the *cidr* property. Moreover, the *DMZ* subnetwork provides the *Bindable* capability for supplying connections to the *DMZ* network. The *www* connectivity is represented by *www\_DMZ\_port*, an instance of the *Port* node type which also includes the *fixed\_ip* property for assigning a fixed address to the connected node. The node *www\_DMZ\_port* is the source of two relationships, namely *BindsTo* and *LinksTo*. The former connects the *Bindable* requirement to the *Bindable* capability of the *Compute* node *www*. The latter connects the *Linkable* requirement to the *Bindable* capability of the *DMZ\_subnet* Subnetwork node.  $\square$

Node instances are collections containing (i) the entity type, (ii) a key-value dictionary of properties, and (iii) a list of requirement bindings. A relationship between two nodes exists when a requirement of a source node instance is bound to the name of the target.

**Example 2.** Consider the YAML specification given in Figure 3.7. It is the TOSCA encoding of the diagram of Figure 3.6. Node *www* (line 1) represents the compute entity for the web server. It is an instance of the `aria.openstack.nodes.Server` type (line 2), i.e., a subtype of `tosca.nodes.Compute` denoting a virtual machine that runs on an OpenStack IaaS.<sup>4</sup> This node contains two properties: the name of the base operating system image (line 4) and the flavor (line 5) of the virtual machine. A port requirement (line 7) permits to establish a relationship

<sup>4</sup>For brevity we may omit namespaces such as `aria.openstack.nodes`.

```

1  www:
2    type: Server
3    properties:
4      image: ubuntu1604
5      flavor: medium
6      requirements:
7        - port: www_DMZ_port
8  www_DMZ_port:
9    type: Port
10   properties:
11     fixed_ip: 198.51.100.5
12   requirements:
13     - network: DMZ
14     - subnet: DMZ_subnet
15  DMZ:
16    type: Network
17  DMZ_subnet:
18    type: Subnet
19    properties:
20      subnet:
21        cidr: 198.51.100.0/24
22    requirements:
23      - network: DMZ

```

Figure 3.7: An excerpt of the TOSCA specification for the working example.

with the `Port` node `www_DMZ_port` (line 8). The port assigns a fixed IP address to the virtual machine using the property *fixed\_ip* (line 11). Also, the port is related to the `DMZ Network` node (line 13) and `DMZ_subnet Subnet` node (line 14). The IP addressing configuration of `DMZ_subnet` is specified in the `subnet` property (line 20).  $\square$

### 3.3.2 SDL types

SDL introduces a number of new node types. Their primary purpose is to model the scenario specific aspects and to integrate them in a TOSCA blueprint. Below we introduce the most relevant ones.

**System types** The type `System` represents the base class of a generic system of the scenario (e.g., workstations, servers, smartphones). Each `System` node is associated (through a relationship) with an existing TOSCA `Compute` node. The `Compute` node is the virtual machine where the `System` runs.

Subtypes of `System` are used to model more specific elements. For instance, `Firewall` represents a `System` node with firewall functionalities. In particular, it is characterized by a default policy, e.g., allows all the traffic passing through its network interfaces (default: allow). It has the capability `Rule` that enables a relationship with the `Policy` nodes. Briefly, a `Policy` node defines a firewall rule through the specification of a traffic pattern. The pattern is represented by the properties `source`, `destination`, `protocol`, and `port`. The meaning is that the

associated firewall has to block the connections matching the pattern.<sup>5</sup>

A `System` node can also be related to some `Artifact`, `Software`, and `User`. An `Artifact` node denotes a file or some other piece of data, e.g., a cryptographic key. A `Software` node represents a program installed on a `System`. It may provide a service endpoint (specifying a port and a protocol) if it is network accessible. Also, a relationship with a `User` defines the running privileges of the software. Finally, a `User` node models a user of the `System` through a requirement `Host`. Its properties include its username, password, and role (denoting its privileges in the `System`).

All the types introduced above include a capability denoting the fact that they can be involved in some vulnerability (see below). For instance, a software can suffer from a known security flaw, while a user can have a weak password.

**Scenario-specific types** The `Vulnerability` type is perhaps the most interesting element in our context. As a matter of fact, it represents a generic, security vulnerability involved in the scenario. Precisely characterize the notion of vulnerability is not trivial and many definitions exist (e.g., see [NIS20]).

In our context, a vulnerability is any security weakness introduced by some (mis) configuration. As discussed above, vulnerabilities may refer to any system type in the scenario. Moreover, they must specify the configuration procedure, i.e., the steps injecting the vulnerability in the scenario, and the exploit operations, i.e., how the attacker uses the vulnerability. Its properties and relationships with the scenario elements vary with the specific vulnerability. Since vulnerabilities are extremely heterogeneous, we do not put further constraints on their structure.

A `Principal` represents a subject operating in the scenario. For instance, attackers (red team) and defenders (blue team) are principals. Typically, a `Principal` has a relationship with some `User` nodes representing the accounts initially controlled. The objective of a `Principal` is to achieve its `Goals`. This is specified through a relationship between the two nodes. A `Goal` represents a state of the scenario that identifies the winning conditions of the related `Principal`, e.g., gain access to a certain system. A detailed discussion on the role of `Goal` nodes is given in Section 3.4.2.

**Relationship types** SDL also introduces new relationship types. The primitive relationship types of TOSCA model the infrastructural dependencies only. For instance, we use `HostedOn` to connect a SDL `System` to the TOSCA `Compute` node hosting it. Scenario-specific dependencies, e.g., the one occurring between a vulnerability and its attack vector, fall outside the scope of TOSCA. In general, one might use the generic `DependsOn` relationship. Yet,

---

<sup>5</sup>Actual firewall policy languages can be more complex and the definition of rigorous languages is still an open research issue, e.g., see [BDF<sup>+</sup>18].

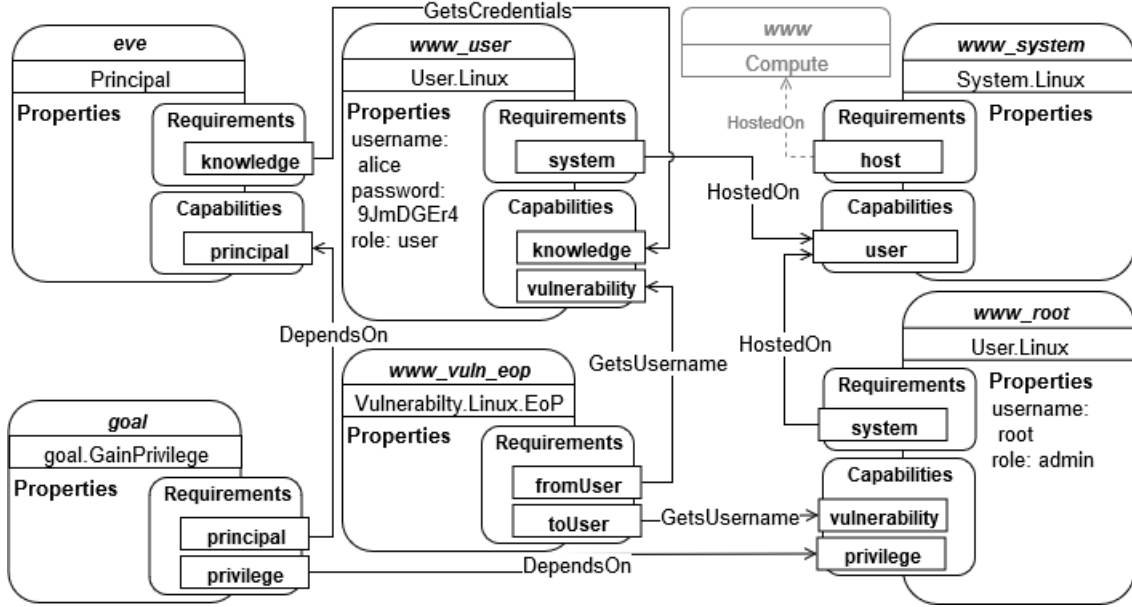


Figure 3.8: A SDL fragment from the specification for the working example.

this would require to customize each instance of the relationship with its specific deployment logic (see Section 3.4.3). An alternative to `DependsOn` is to define new relationship types that commonly occur in the scenarios. For instance, a relationship `GetsUsername` can connect a `Vulnerability` node to a `User` node. The meaning is that the `Vulnerability` node reads the `username` property of the `User` node and uses it for the configuration of the vulnerability. To exemplify, think of a user enumeration vulnerability where an attacker enumerates the users of a service by testing a dictionary of common usernames. To properly configure it, one has to make sure that the target username belongs to the dictionary.

**Example 3.** The SDL diagram in Figure 3.8 extends the TOSCA specification of Example 1 by adding some elements of Scenario 1 (see Section 3.1). In particular, here we only consider the third, EoP vulnerability of Scenario 1 (see Section 3.1) and we assume that the red team already knows the credentials to remotely access as an unprivileged user. The `www_system` node represents the Linux system running on the `Compute` node `www`. The `User.Linux` node `www_user` is hosted on the Linux system. The properties of `www_user` define its username, password, and role (where `user` stands for a standard, unprivileged user). Similarly, `www_root` models the administrator of `www_system`. Like `www_user`, `www_root` has a username and a role, i.e., `admin`. However, by not setting the `password` property, we model the passwordless,

```

1  User:
2    derived_from: Root
3    properties:
4      username:
5        type: string
6        required: yes
7      password:
8        type: string
9      role:
10       type: string
11       default: user
12       constraints:
13         - valid_values: [admin, user]
14     capabilities:
15       vulnerability:
16         type: VulnerabilityContainer
17       knowledge:
18         type: KnowledgeProvider
19     requirements:
20     [...]
21 Principal:
22   derived_from: Root
23   properties:
24     name:
25       type: string
26       required: yes
27   requirements:
28     - knowledge:
29       capability: KnowledgeProvider
30   capabilities:
31     principal:
32       type: PrincipalProvider
33 User.Linux:
34   derived_from: User
35   [...]
36 Vulnerability.Linux.EoP:
37   derived_from:
38     Vulnerability.Linux
39   requirements:
40     - fromUser:
41       capability:
42         PrivilegeProvider
43       relationship:
44         GetsUsername
45       node: User.Linux
46     - toUser:
47       capability:
48         PrivilegeProvider
49       relationship:
50         GetsUsername
51       node: User.Linux
52   [...]
53 Goal.GainPrivilege:
54   derived_from: Goal
55   properties:
56     description:
57       type: string
58   requirements:
59     - principal:
60       capability: PrincipalProvider
61       node: Principal
62     - privilege:
63       PrivilegeProvider
64       node: User

```

Figure 3.9: An excerpt of SDL node types declaration.

default configuration of the Ubuntu Linux distribution [Can19] (represented by the `www` node in Example 1).

The `www_vuln_eop` node represents the EoP vulnerability mentioned above. Its type is `Vulnerability.Linux.EoP` and it has two requirements, i.e., `fromUser` and `toUser`. Respectively, they connect to the unprivileged and the privileged users. Such a connection occurs through the `GetsUsername` relationship.

A `Principal` node `eve` represents the attacker (red team). It is related to `www_user`, i.e., the account initially controlled by the attacker. Finally, the `goal` node is related to `eve` and it represents the objective of the attacker. It is an instance of the `goal.GainPrivilege` type and it has two requirements, namely `principal` and `privilege`, meaning that the connected `Principal` (namely `eve`) aims at acquiring the privileges of the connected `User` (namely `www_root`). Notice that this is not the final goal of Scenario 1 (i.e., reading the content of the database). In fact, it is an intermediate goal, i.e., a sub-goal enabling the final one, that we consider here for the sake of presentation.  $\square$

**Example 4.** In Figure 3.9 we give the type declaration of the node types related to the vulnerability introduced in Example 3. The `User` type (lines 1-20) represents a SDL primitive and inherits from the root of all the SDL types, namely `sdl.nodes.Root`. It has three properties: `username` (line 4), is a mandatory string identifying the user in the system, `password` (line 7), an optional string representing the user's password, and `role` (9) representing the user's privileges. Two roles are modeled in the example, i.e., `admin` and `user` (line 12-13). An unspecified value for the `role` property implies a default user role (line 11). Moreover, `User` has the capability `vulnerability` (line 15) of type `VulnerabilityContainer` since a vulnerability may affect it.

`Principal` (lines 21-32) represents another SDL primitive type. It has a mandatory `name` property (line 24) for identifying the principal. A `knowledge` (line 28) requirement to link the principal with her initial knowledge, e.g., the controlled account `www_user` in Figure 3.8. Moreover, `Principal` has the capability `principal` (line 31) of type `PrincipalProvider` discussed in Example 3.

The type `User.Linux` (lines 33-35), extends the `User` type described above. The definition of a subtype for `User` is necessary to discriminate between OS-specific procedures such as user creation and vulnerability exploitation.

The type `Vulnerability.Linux.EoP` (lines 36-52) extends the supertype `Vulnerability.Linux`. It contains the `fromUser` and `toUser` requirements described in Example 3. They both specify `VulnerabilityProvider` as capability and `User.Linux` as target node type. Also, they both only admit a relationship of type `GetsUsername`.

The `Goal.GainPrivilege` type (lines 53-64) inherits from the `Goal` primitive type. It has a `description` (line 56) property that we use to annotate each instance of the goal. For instance,



it may describe the goal w.r.t. the current scenario, e.g., for team briefing. As anticipated in Example 3, this type is related with a `Principal` and a `User`. These two relationships are enabled by the requirements `principal` (line 59) and `privilege` (line 62), respectively.  $\square$

### 3.3.3 Behavior and runtime

The scenario story line amounts to a sequence of actions that make the scenario evolve over time. These actions play a crucial role for the verification and testing of the scenario. Thus, we need to introduce two distinct abstractions, one for statically modeling an action and one for providing its dynamic semantics. For this reason, all the SDL node and relationship types have two special properties, i.e., `behavior` and `runtime`. Intuitively, `behavior` properties contain terms used for the verification process (see Section 3.4.2). Instead, `runtime` is associated with *commands*, for instance shell scripts, to be executed. Such commands are mostly used for the testing phase (see Section 3.4.3). Both runtime and behavior properties are given as finite mappings between unique identifiers and commands and terms, respectively. Moreover, we require the two mappings to have exactly the same domain. Differently said, the runtime maps an identifier to a command if and only if the behavior maps the same identifier to a term.

### 3.3.4 Access pattern language

TOSCA natively provides operations, called intrinsic functions, to access the information stored inside a node [RBL19, § 4.3]. For instance, a node  $n$  can use `get_property: [r, p]` to read the value assigned to property  $p$  by the node related to  $n$  through the requirement  $r$ . Notice that intrinsic functions can only walk through a single relationship. This is reasonable for the design of an infrastructure, where each node is related to the others it depends on. However, it makes extremely hard to model complex dependencies such as those introduced by the `behavior` property of vulnerabilities and goals. The motivation is that, for instance, a goal can be related to nodes that are very far in the blueprint.

For this reason, we introduce an *access pattern* language to specify structured, path-based queries. An access pattern  $\rho$  follows the syntax below.

$$\begin{aligned}
\rho &::= \pi[P] \mid \pi\{A\} \\
\pi &::= \pi_{nod} \mid \pi_{rel} \mid \pi_{cap} \\
\pi_{nod} &::= \text{this} \mid \pi_{rel}.\text{src} \mid \pi_{cap}.\text{node} \\
\pi_{cap} &::= \text{this} \mid \pi_{nod} \leftarrow C \mid \pi_{rel}.\text{dst} \\
\pi_{rel} &::= \text{this} \mid \pi_{nod} \rightarrow R \mid \pi_{cap}.\text{rel}
\end{aligned}$$

An interpreter evaluates and replaces an access pattern  $\rho$  with a set<sup>6</sup> of values according to the target SDL specification. In particular,  $\pi[P]$  amounts to the value of property  $P$  of the SDL elements *pointed* by  $\pi$  (therefore called a pointer).<sup>7</sup> Similarly,  $\pi\{A\}$  reduces to the value of the attribute  $A$  of the elements pointed by  $\pi$ . A pointer  $\pi$  can be of three kinds depending on the class of the SDL elements it refers to, i.e., nodes ( $\pi_{nod}$ ), relationships ( $\pi_{rel}$ ) or capabilities ( $\pi_{cap}$ ).<sup>8</sup> Each element can use `this` as a self-pointer.<sup>9</sup> A node pointer  $\pi_{nod}$  is also obtained by means of the operator `.node` applied to (a pointer to) a capability contained by the node. A pointer to a capability  $C$  is obtained by applying the operator `<-` to a node pointer. Moreover, from a relationship pointer  $\pi_{rel}$  one can access the destination capability by means of the operator `.dst`. Conversely, a pointer to a relationship can be obtained either (i) from a node pointer  $\pi_{nod}$  by means of the requirement  $R$  where the relationship originates (operator `->R`) or (ii) from a capability through the operator `.rel`. For the sake of presentation we also introduce the following abbreviations.

$$\pi_{nod}=>R \equiv \pi_{nod}->R.dst.node$$

$$\pi_{nod}<=C \equiv \pi_{nod}<-C.rel.src$$

**Example 5.** Consider the following access patterns defined by node `eve` (see Figure 3.8).

```
this[name]    this->knowledge[name]    this=>knowledge[role]
```

The first pattern trivially evaluates to `eve`. The second one requires to access property `name` of the relationship originating from the requirement `knowledge`, i.e., `GetsCredentials`. Finally, the last access pattern follows the relationship originating from `knowledge` and points to node `alice` whose property `role` is assigned to `user`.  $\square$

## 3.4 Introducing CRACK

In this section, we present our framework CRACK. We start in Section 3.4.1 by providing a general description of its structure. Then we detail the main constituents, i.e., its specification language, the scenario verification and automatic testing process.

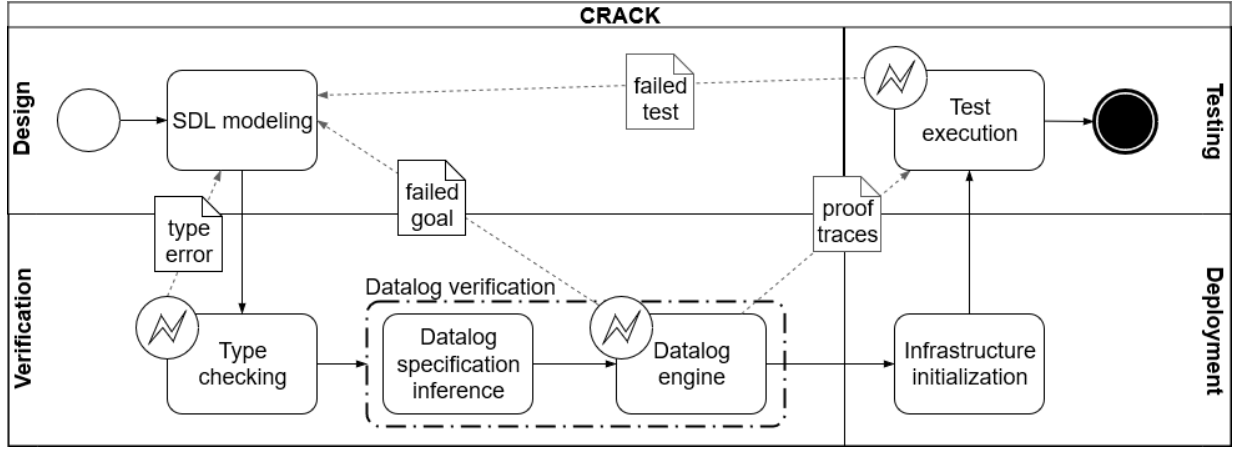


Figure 3.10: CRACK general workflow.

### 3.4.1 Overview of the approach

Figure 4.9 depicts the abstract workflow of CRACK. The scenario development workflow starts with the modeling task. During this task, the designer creates a blueprint of the scenario by using SDL (see Section 3.3). The model is then type checked to detect possible inconsistencies. If it is the case, type errors are returned to the modeling task to be fixed. Otherwise, the process proceeds to the verification task. In the current implementation, CRACK generates a Datalog specification from the model and feeds it to a Datalog engine (see Section 3.4.2). We stress the fact that our approach can be extended with other verification techniques. To support this operation, CRACK is based on a modular design. If the Datalog verification fails, the unreachable goals are returned to the design process, otherwise a proof trace for each goal is generated and the process moves to the infrastructure initialization task. When the infrastructure is up and running, CRACK performs the test execution phase (described in Section 3.4.3). This task converts the proof traces into unit tests and executes them on the deployed infrastructure. If a test fails, a feedback is provided to the user in order to refine the scenario blueprint. Eventually, when all the tests are passed, the scenario is ready to be played.

<sup>6</sup>Since SDL relationships can be many-to-many, the evaluation of an access pattern is not guaranteed to result in a single value.

<sup>7</sup>For brevity, we may omit  $[P]$  when  $P = \text{name}$ .

<sup>8</sup>Notice that requirements cannot declare properties in TOSCA.

<sup>9</sup>We omit it when clear from the context.

Table 3.1: Predefined Datalog predicates (excerpt).

Predicate	Description
<code>knows (A, D)</code>	Principal A knows datum D (e.g., a password)
<code>hasAccount (A, H, U)</code>	Principal A has an account on host H as user U
<code>hasUser (U, H, P, R)</code>	Host H has user U with password P and role R
<code>listeningOn (H, Q, S)</code>	Host H has a software on port S with protocol Q
<code>isConnected (H, N)</code>	Host H is connected to network N
<code>hostACL (H, K, Q, S)</code>	Host H can access host K via protocol Q on port S

### 3.4.2 Scenario Verification

SDL inherits the TOSCA type system (cf. Section 3.3.1). A well-typed blueprint enjoys some properties of interest, such as the coherence between nodes and relationships [BTS17]. Although efficient, type checking cannot verify more complex properties. To overcome this limitation, we introduce a further verification phase that converts a well-typed SDL model into a Datalog specification (see Section 3.4.2). Objectives are then encoded as queries that must be satisfied by the specification.

A successful verification yields as a set of proof traces, one for each objective. Proof traces are later used as input to the deployment and testing phases (cf. Section 3.4.3). Interestingly, a verification failure is also useful: invalid queries can be productively used (e.g., see [KLS12]) to identify bugs in the model that originated the specification.

#### Encoding

The encoding process generates a Datalog specification from a scenario blueprint. The Datalog terms refer to a set of predefined predicates. Some predicates are inspired by [OGA05]. The most relevant ones are listed in Table 3.1.

Facts and clauses belong to three blocks, i.e., *constants*, *behaviors* and *goals*. Constants include Datalog terms that model the standard behavior common to any infrastructure. For instance, the clause

```
hasAccount (A, H, U) :- hasUser (U, H, P, R), knows (A, U), knows (A, P) .
```

means that principal A owns account U on host H if user U exists on H and A knows both U and the associated password P.

Behaviors blocks contains the terms introduced by the SDL elements through the behavior property (see Section 3.3). The behavior property consists of a mapping between identifiers and

*term patterns*. A term pattern resembles a standard Datalog term, i.e., a fact or a clause, but its parameters can also be access patterns (see Section 3.3.4).

**Example 6.** Consider the SDL fragment of Example 3. We show the Datalog term generated by the EoP vulnerability. Let assume that `www_vuln1` has the following behavior property.

```
hasAccount(A, H, =>ToUser) :- hasUser(=>ToUser, H, P1, R1),
                               hasUser(=>FromUser, H, P2, R2),
                               hasAccount(A, H, =>FromUser).
```

The term above models the vulnerability prerequisites (clause premises) and effect. In practice, the vulnerability allows a principal `A` to obtain the control over a target high-privileged user `=>ToUser`<sup>10</sup> by leveraging a misconfiguration. In particular, the misconfiguration has to do with a low-privileged user `=>FromUser` that can impersonate `=>ToUser` when launching a certain command. For this to happen three conditions must be satisfied.

- i)* The high-privileged account `=>ToUser` exists on the target system.
- ii)* A low-privileged account `=>FromUser` also exists on the target system.
- iii)* Principal `A` has control over the low-privileged account.

Similarly, the behavior property of the relationship `GetsCredentials` is defined as follows.

```
knows(.src, .dst.node[password])
knows(.src, .dst.node[username])
```

Finally, also the nodes of type `User.Linux`, i.e., `alice` and `root`, define a term pattern in their behavior property.

```
hasUser(this[username], =>System, this[password], this[role])
```

According to the specification of Example 3, all the above term patterns contribute to the following Datalog specification.

```
/* Constants */
hasAccount(A,H,U) :- hasUser(U,H,P,R), knows(A,U), knows(A,P).
/* EoP vulnerability */
hasAccount(A, H, root) :- hasUser(root, H, P1, R1),
                          hasUser(alice, H, P2, R2),
                          hasAccount(A, H, alice).

/* GetsCredentials */
knows(eve, 9JmDGEr4).
knows(eve, alice).
```

---

<sup>10</sup>Recall that this is an abbreviation for `this=>ToUser[name]`.

```

/* root */
hasUser(root, www_system, , admin).
/* alice */
hasUser(alice, www_system, 9JmDGEr4, user).

```

□

Finally, the SDL goals result in queries to be evaluated against the Datalog model. Such queries denote that a certain configuration is reachable in the scenario.

**Example 7.** Consider the node goal of Example 3. Its behavior property contains the term `hasAccount(=>Principal, =>Privilege=>System, =>Privilege)?` which reduces to the query `hasAccount(eve, www_system, root)?`.

Notice that, in this particular example, the goal query contains no free variables. Thus, its evaluation results in a plain boolean value. □

## Verification

The verification process boils down to running a Datalog engine against the goal queries. The verification fails when one or more queries cannot be satisfied. In such a case, the failure denotes that a principal cannot achieve one of its goals.

**Example 8.** Consider again the Datalog specification of Example 6 and the query of Example 7. The query is trivially satisfied by the specification. The generated proof trace is as follows.

```

1. hasUser(root, www_system, , admin).           /* Fact */
2. hasUser(alice, www_system, 9JmDGEr4, user).    /* Fact */
3. knows(eve, alice).                             /* Fact */
4. knows(eve, 9JmDGEr4).                          /* Fact */
5. hasAccount(eve, www_system, alice) :-
    hasUser(alice, www_system, 9JmDGEr4, user),   /* From 2 */
    knows(eve, alice),                           /* From 3 */
    knows(eve, 9JmDGEr4).                         /* From 4 */
6. hasAccount(eve, www_system, root) :-
    hasUser(root, www_system, , admin),           /* From 1 */
    hasUser(alice, www_system, 9JmDGEr4, user),   /* From 2 */
    hasAccount(eve, www_system, alice).           /* From 5 */

```

We discuss the steps of the proof trace in backward order. The last step (6) concludes the proof by inferring the goal query. The proof step consists of an application of the clause introduced by the EoP vulnerability (see Example 6). To apply the clause, three preconditions must be satisfied. Two of them amount to facts appearing in the specification (i.e., 1 and 2), thus requiring no further proofs. Instead, the last one is inferred by applying the clause of the constants block (see

Example 5). The inference step (5) is based on three premises, i.e., 2, 3 and 4. Since all of them are facts appearing in the specification, the proof is completed.  $\square$

### 3.4.3 Deployment and testing

Once verified, a blueprint can be instantiated and executed. In general, the deployment process is not guaranteed to preserve the model-verified properties. As a matter of fact, the abstract, high-level design purposely neglects some implementation aspects that may affect the scenario at runtime. For instance, a piece of software may behave differently from the expectations of the scenario designer.

To favor a prompt detection and debugging of the scenario, we leverage the verification proof traces to automatically generate and run tests. Each test aims at confirming that a model-verified property also holds in the deployed scenario.

#### Deployment

The deployment phase generates the directives for the IaaS provider. Many solutions exist for the interpretation/translation of the TOSCA specifications into the orchestration instructions of the major IaaS provider (e.g., see Cloudify [Clo19], ARIA TOSCA [ARI19], OpenTOSCA [BBH<sup>+</sup>13], Alien4Cloud [Ato19] and Heat-Translator [Ope19b]). All of them only apply to the standard TOSCA. Clearly, to support our SDL we could customize one of them. However, this would make SDL incompatible with the other existing TOSCA implementations.

To avoid the customization and preserve the compatibility with the existing TOSCA-based technologies, we rely on the TOSCA interfaces (see Section 3.3.1). In particular, for all the SDL node types we declare a standard interface. Recall that all of the SDL types have some relationship (either direct or indirect, i.e., through a relationships path) with one TOSCA Compute node (see Section 3.3.1). Such relationship is resolved at runtime to identify the platform where the interface task must be executed. Thus, all the tasks defined by our standard interfaces result in a configuration command to be executed on a certain TOSCA Compute node. To clarify we propose the following example.

**Example 9.** Consider again the EoP vulnerability (node `www_vuln1`) of Example 3. Figure 3.11 shows the continuation of the declaration of `linux.vulnerability.EoP` (given in Figure 3.9). Clearly, the vulnerability must be enabled by properly configuring `www`, i.e., the Compute node where the privilege escalation takes place. In this case, the node `www` is identified as the system hosting the User nodes (namely, `fromUser` and `toUser`) related to the vulnerability.

```
[...]
38:  interfaces:
39:    standard:
40:    configure:
41:    implementation: /scripts/eop-apt-config.sh
```

Figure 3.11: Extended YAML declaration of the EoP vulnerability.

```
#!/bin/bash

# read username attributes
fromUser= #...
toUser= #...

echo "$fromUser ALL=($toUser) NOPASSWD: /usr/bin/apt-get" >> /etc/sudoers
```

Figure 3.12: Implementation of `eop-apt-config.sh`.

The implementation of the interface is given through a shell script (line 41). We report core operations of the script in Figure 3.12. Briefly, the script enables a specific implementation of the EoP vulnerability for Debian-based OS. The vulnerability is activated through a misconfiguration of the `/etc/sudoers` file that allows underprivileged users to invoke the Debian package manager `apt-get` in passwordless mode [The20f]. In particular, the script retrieves the usernames of the two users. Usernames are obtained by reading the corresponding SDL attributes which contain actual, runtime values. Then, the vulnerability is enabled by appending the vulnerable configuration line to `/etc/sudoers`. □

## Testing

The test execution process consists of translating a Datalog proof trace into an executable test and run it on the deployed scenario. The testing process is handled by a *Test Execution Engine* (TEE).

Figure 3.13 schematically depicts the TEE and its relationship with a running scenario. The test execution proceeds in this way. The next fact  $F(\bar{v})$ , namely the *Fact Under Test* (FUT), in the input Datalog trace is extracted and given to a *trace interpreter*. The interpreter retrieves a *test driver*, i.e., a script specifically designed to test facts referring to the predicate  $F$ , from an internal database. The test driver has a predefined interface and it may refer to values taken from either the FUT, the scenario blueprint or runtime information generated by the test execution and stored in a *test database*. The structure of the test database is straightforward. In particular, it consists of a table for each Datalog predicate where each column corresponds to a parameter.



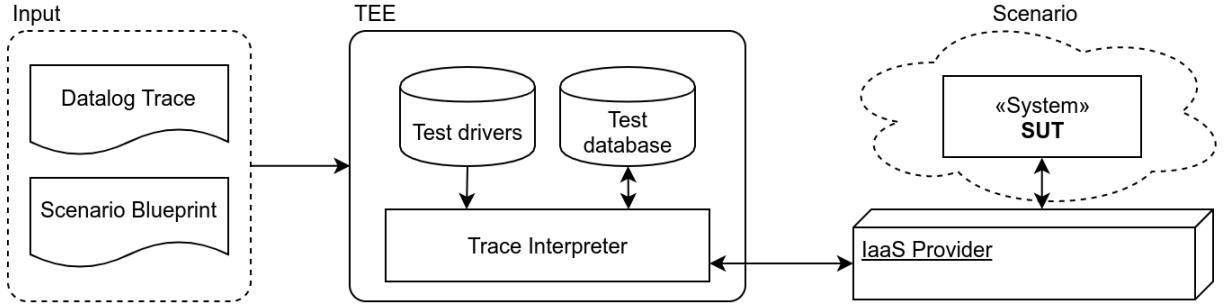


Figure 3.13: The Test Execution Engine of CRACK.

```

 $\bar{u}$  := get_values_from_fut()
 $\bar{v}$  := get_values_from_blueprint()
 $\bar{w}$  := get_values_from_testdb()

sut := get_sut_from_blueprint()
test := get_runtime_from_blueprint(sut, fut)
insert_values_in_script(test,  $\bar{u}$ ,  $\bar{v}$ ,  $\bar{w}$ )
r := submit_script_to_iaas(sut, test)

if(is_not_successful(r)) then test_failed()
else insert_values_in_testdb(r) and test_passed()

```

Figure 3.14: Test driver pseudo-code.

Notice, however, that actual, runtime values may be considered when they simplify the testing operations. For instance, one might prefer to store the actual username of a user, rather than its SDL identifier as it appears in the Datalog trace (see Example 10).

The general structure of a test driver is given in Figure 3.14. Each driver starts by retrieving the necessary data from the FUT ( $\bar{u}$ ), the blueprint ( $\bar{v}$ ) and the test database ( $\bar{w}$ ). Then it identifies the *system under testing* (SUT) by checking which SDL node declares the FUT in its behavior. Similarly, the test driver retrieves the corresponding *test* script from the runtime property of the SUT. Before running the script, the actual test values must be inserted, i.e., passed as the input parameters of the script. Eventually, the script is submitted to the IaaS provider that executes it on the SUT and returns the output values. The output is a tuple on which a successful condition can be checked. If the check is not passed the test fails, otherwise the driver inserts the output values in the test database and terminates.

**Example 10.** Consider again the EoP vulnerability of our working example. Its *runtime* property

```

1. #!/bin/bash
2. # Test driver for hasAccount(principal, host, user)

3. function getParam() {
4.   # parses and reads the input tuples  $\bar{u}, \bar{v}, \bar{w}$ 
5. }

6. principal=$(getParam $1) # the principal
7. host=$(getParam $2)     # the host
8. username=$(getParam $3) # user's username
9. knowledge=$(getParam $4) # principal's knowledge

10. r=""
11. for k in $knowledge
12. do
13.   if [ "$(grep -P "^$k.+NOPASSWD.+apt.+" /etc/sudoers)" ]; then
14.     r=$(su $k -c 'sudo /usr/bin/apt-get update \
15.       -o APT::Update::Pre-Invoke::="id"' | grep $username) && break
16.   done

17. if [ ! "$r" ]; then
18.   username=""
19. fi

20. echo "($principal, $host, $username)"
21. # is_not_successful() iff $username == ""

```

Figure 3.15: Runtime script used to test the EoP vulnerability.

contains the script of Figure 3.15.

The script starts by reading the inputs provided by the test driver (lines 6–9). For this purpose a utility function is used (getParam, line 3). There are four inputs passed by the driver. Two of them, i.e., principal and host, are taken from the FAU. Instead, username is taken from the corresponding attribute of the node User. The reason is that the script requires the actual username, rather than the User node identifier (which appears in the proof trace). The last parameter is the knowledge of the principal, i.e., the content of table Knowledge (for the current principal) in the test database. All in all, the script checks if the principal's knowledge contains a username that enables the exploitation, i.e., escalating the privileges. In details, the main body of the script amounts to a for loop iterating on each element  $k$  of knowledge (line 11). Then, through a regular expression matching (line 13), the script checks if the configuration file /etc/sudoers contains a line (i) starting with (username)  $k$ , (ii) containing the keyword NOPASSWD, i.e., indicating the passwordless command execution mode, and (iii) containing the exploitable command apt (see Example 9). If the match occurs, the script executes the exploit (line 14) as the user  $k$  (su  $k$  -c). The EoP exploit leverages a configuration op-

tion (`APT::Update::Pre-Invoke::`) for invoking arbitrary commands before updating the `apt` package index file. In particular, the script invokes the command `id` for printing the username of the current user. If the output matches `username`, i.e., the privileged user, the for loop breaks and `r` is assigned to the command output (lines 14–15). After the loop, `r` is left empty only if the script failed in running the exploit. Hence, the script overwrites `username` with an empty string (line 17-19). Eventually, the script returns the tuple (`principal`, `host`, `username`) (line 20). The driver checks whether the script failed by comparing `username` with the empty string.  $\square$

A test is successful when each step succeeds. In this case, we obtain an evidence that the proof trace has been preserved after the scenario deployment. Otherwise, we get a useful indication of what went wrong. In particular, a test failure amounts to the failure of a certain script. By reversing our mapping, we find which clause, appearing in the proof trace, is not satisfied by the deployed scenario. As a consequence, the blueprint can be inspected to understand and fix the error. When all the tests are successful the scenario is ready.

### 3.5 CRACK Demo

In this section, we provide a demonstration of CRACK applied to our working example. CRACK is available as a free open source software on GitHub [Rus20a]. More precisely, the version used for this work is CRACK v1.0.1 [Rus20b]. The repository contains (i) the source code, (ii) the configurations and (iii) the library of SDL elements required for replicating the experiments described below. In particular, the SDL implementation consists of 62 node types, 20 capabilities and 11 relationships. All together, they amount to 4343 YAML lines of code (loc) for the type definitions, 916 shell loc for the deployment interfaces and 476 shell loc for the runtime scripts.

CRACK is built on top of the Apache ARIA project (see Section 3.4.3). In particular, we rely on ARIA for supporting the design (see Section 3.3) and deployment (see Section 3.4.3) phases. Instead, the verification (see Section 3.4.2) and testing (see Section 3.4.3) phases are enabled through a plugin extension of ARIA. Moreover, we use pyDatalog [Car16] as the Datalog engine for the verification module.

Our testing environment runs on an Ubuntu Linux, version 16.04.5 LTS, installed on a two Intel Xeon Processors E5440 server with 32GB of RAM. Finally, we use DevStack [Ope19a] for installing OpenStack 3.16.0 (Rocky).

The outline of the demonstration follows. We start from the design of the first scenario of our case study in Section 3.5.1. Then, we verify (Section 3.5.2), deploy and test (Section 3.5.3) the scenario. In Section 3.5.4, we simulate the red team activity on the running scenario. Also, there we evaluate the effort for migrating between the first and the second scenario of the case study.

```

Scenario1      # Scenario root directory
├── Types      # SDL types directory
│   └── ...
├── scenario.yaml # main specification file
├── DMZ        # DMZ specifications directory
│   ├── ns.yaml # ns host specification
│   ├── www.yaml # www host specification
│   └── ...
├── Server     # Server specifications directory
│   └── db.yaml # db host specification
├── Internet   # Internet specifications directory
│   ├── provider.yaml # provider router specification
│   ├── client.yaml  # client host specification
│   ├── root-ns.yaml  # root-ns host specification
│   └── ...
├── IoT        # IoT specifications directory
│   └── ...
├── config.yaml # input variables
├── network.yaml # network infrastructure specification
└── firewall.yaml # firewall specification

```

Figure 3.16: The file tree for the Scenario 1.

### 3.5.1 Design

For Scenario 1, we carry out the design from scratch. That is, we define all the elements without assuming any prior scenario design. The design process starts from the theater elements, e.g., networks and compute nodes, and incrementally proceeds to the scenario aspects, e.g., vulnerabilities. The entire scenario is encoded in 2385 loc organized in 29 YAML files.

The files are structured as depicted in Figure 3.16. Type specifications are placed in a dedicated folder `Types`. The file `scenario.yaml` is the entry point for the scenario specification. This file imports all the other YAML specifications and contains the declarations of the scenario principals and goals. In particular, recall that in our scenarios we have only one goal for `eve` (a.k.a. the red team), i.e., exfiltrating data from `db`. We use the label `DB_confidential` to denote the target data. As a consequence, the goal amounts to `knows('eve', 'DB_confidential')`. The network infrastructure is defined in `network.yaml`. For each network, e.g., `DMZ`, a directory is used to contain the relevant specifications. Each host is defined in a separate YAML file. A host file contains the definition of the corresponding Compute node and its configuration, e.g., `www.yaml`. Finally, we introduce a utility file `config.yaml`. Such a file contains values assigned to *common reconfigurable* properties, e.g., network addresses, domain names, and

usernames. These properties can be modified for quickly reshaping the scenario by acting on a single file.

Figure 3.17 shows the YAML specification of `www` for the first scenario of the working example. We start by defining the OpenStack Compute node running `www` and its connection to the DMZ subnetwork via `www_DMZ_port` (lines 1 and 8, respectively). Then, we connect `www_system` (line 15) to `www`. As stated in Section 3.3.2, `www_system` enables the relationships with the SDL nodes. For instance, the ssh server `www_ssh` (line 25), the Apache HTTP server [Apa20] `www_http` (line 31), the Apache module for running PHP [The20h] pages `www_php` (line 38), and WordPress [Wor20] `www_cms` (line 43) are software components running on `www`. Also, we define three users, i.e., `www_root` (line 71), the `www_http_user` (line 79) and `www_user` (line 86). Finally, we add one (mis)configuration and three vulnerabilities. Briefly, they implement three (number 2, 3 and 4) of the vulnerabilities introduced in Section 3.1.

2. The configuration `www_http_userdir` (line 95) exposes the users home directories and the vulnerability `www_weak_enumerable` (line 103) modifies the usernames to ensure that they are enumerable (in the sense explained in Section 3.1).
3. The node `www_vuln_weakpass` (line 110) configures the weak password for `www_user`.
4. The node `www_vuln_eop` (line 119) injects the EoP vulnerability (see Example 9).

Notice that, vulnerability 1. does not appear in `www.yaml` as it affects `db`.

At the end of the design step, we submit `we.yaml` to ARIA. This operation includes the type checking step (see Section 3.4.2). When the operation terminates, the scenario, called a *service* in the ARIA terminology, is saved as `we_service`. Although the scenario is technically deployable, we still have to verify it with CRACK.

### 3.5.2 Verification

CRACK provides a TOSCA workflow, called *verify*, that implements the verification procedure described in Section 3.4.2. We invoke the verify workflow through the ARIA workflow execution engine. Figure 3.18 shows the output for the first scenario of the working example. The generated Datalog specification is saved as `datalog.py` using the pyDatalog format.

Since the test goal is verified (`Result: TRUE`), the `goal1.trace` file containing the Datalog proof trace is generated. Also, the workflow creates `datalog.json`. Briefly, it contains a mapping between the Datalog terms and the SDL type declaring them. Such a mapping binds each FUT appearing in the proof trace with the corresponding SUT during the test execution process (see Section 3.4.3). The mapping is stored as a list of records of the form

```

1  www:
2  type: aria.openstack.nodes.Server
3  properties: # [...]
4  requirements:
5  - key_pair: keypair
6  - port: www_DMZ_port
7
8  www_DMZ_port:
9  type: aria.openstack.nodes.Port
10 properties: # [...]
11 requirements:
12 - subnet: DMZ_subnet
13 - network: DMZ
14
15 www_system:
16 type: sdl.nodes.System.Linux
17 properties:
18 hostname: {get_input: www_hostname}
19 domain: {get_input: www_domain}
20 dns: {get_input: www_dns}
21 requirements:
22 - host: www
23 - dependency: Firewall_InitPermissions
24
25 www_ssh:
26 type: sdl.nodes.Software.Server.
27 SSH.Linux.OpenSSH
28 requirements:
29 - swcontainer: www_system
30
31 www_http:
32 type: sdl.nodes.Software.
33 Server.HTTP.Linux.Apache
34 requirements:
35 - swcontainer: www_system
36 - user: www_sysuser1
37
38 www_php:
39 type: sdl.nodes.Configuration.
40 HTTP.Linux.Php.Apache
41 requirements:
42 - server: www_http
43
44 www_cms:
45 type: sdl.nodes.Software.
46 Server.CMS.Linux.Wordpress
47 properties:
48 db_username: root
49 db_password:
50 {get_input: db_admin_pass}
51 db_host:
52 {get_input: db_server_ip}
53 db_name:
54 {get_input: www_cms_dbname}
55 url: {concat:
56 [get_input: www_hostname,
57 ' ', get_input: www_domain]}
58 title:
59 {get_input: www_cms_title}
60 admin_user:
61 {get_input: www_cms_login}
62 admin_password: {get_input:
63 www_cms_password}
64 admin_email: {concat:
65 ['webmaster@',
66 get_input: www_domain]}
67 requirements:
68 - container: www_http
69 - db: db_cmsdb
70 - dependency: db_config
71
72 www_root:
73 type: sdl.nodes.User.Linux
74 properties:
75 username: root
76 role: admin
77 requirements:
78 - system: www_system
79
80 www_http_user:
81 type: sdl.nodes.User.Linux
82 properties:
83 username: www-data
84 role: user
85 requirements:
86 - system: www_system
87
88 www_user:
89 type: sdl.nodes.User.Linux
90 properties:
91 username: alice
92 password: 9JmDGEr4
93 role: user
94 requirements:
95 - system: www_system
96
97 www_http_userdir:
98 type: sdl.nodes.Configuration.
99 HTTP.Linux.UserDir.Apache
100 requirements:
101 - user: www_user1
102 - server: www_http
103 - dependency: www_weak1
104
105 www_weak_enumerable:
106 type: sdl.nodes.Vulnerability.
107 Linux.User.RemoteEnumerable
108 requirements:
109 - user: www_user
110 - server: www_http
111
112 www_vuln_weakpass:
113 type: sdl.nodes.Vulnerability.
114 Linux.User.RemoteWeakPassword
115 requirements:
116 - user: www_user
117 - server: www_ssh
118 - dependency:
119 www_weak_enumerable
120
121 www_vuln_eop:
122 type: sdl.nodes.Vulnerability.
123 Linux.EOP.APT
124 requirements:
125 - fromUser: www_user
126 - toUser: www_root
127 - dependency:
128 www_weak_enumerable

```

Figure 3.17: The content of `www.yaml`.

```

Terminal
File Edit View Search Terminal Help
~/git/SDL/scenarios$ aria execution start verify -s we_service
- Test goal 1: eve knows DB_confidential [print(knows('eve_1', 'DB_confidential'))]
  Result: TRUE
  Save trace in: /home/enriquez/git/SDL/scenarios/output/we1//goal1.trace
Starting execution. Press Ctrl+C cancel
Starting 'verify' workflow execution
'verify' workflow execution succeeded
Execution has ended with "succeeded" status
~/git/SDL/scenarios$ ls -1 ~/git/SDL/scenarios/output/we1/
datalog.json
datalog.py
goal1.trace
~/git/SDL/scenarios$

```

Figure 3.18: The execution of the *verify* workflow.

```

Nat : { "node": SDL node identifier,
        "type": SDL node type,
        "key" : Behavior/runtime identifier }

```

where *Nat* is a unique natural number, *node* is the name of a node in the blueprint, *type* is the SDL type of the node and *key* denotes a valid entry in the behavior/runtime mapping of the node (see Section 3.3.3).

Figure 3.19 shows an excerpt of the `datalog.py` file. Briefly, it amounts to the Datalog facts and clauses generated for *www*. The syntax follows the `pyDatalog` format where facts are preceded by the '+' symbol and clauses use '<=' and '&' in place of ':' and ',' (respectively, cf. Section 3.2.3). Furthermore, notice that each Datalog predicate has an extra argument, i.e., the first one, being a natural number. Such an argument is assigned to a unique constant in each fact and left term of the clauses. Instead, the argument appears as an unconstrained variable in the premises of each clause. All in all, this argument maps each step of a proof trace to a corresponding entry in the `datalog.json` file (through the values *Nat* discussed above).

Figure 3.20 contains a fragment of the proof trace generated by the *validate* workflow. For the sake of presentation, we omit the proof details and we only report the facts proved at each step by the `pyDatalog` deduction system. The proof succeeds by achieving the goal, i.e., `knows(21, 'eve', 'DB_confidential')`, at the final step.

### 3.5.3 Deployment and testing

The deployment process results in the infrastructure depicted in Figure 3.21. The infrastructure contains the components discussed in Section 3.1. It is worth noticing that the actual topology

```

1. + isConnected(1, 'www', 'DMZ_subnet')
2. + hasUser(2, 'www_root', 'www', 'None', 'admin')
3. + listeningOn(3, 'www', 'tcp', '22')
4. hasAccount(4, A, 'www', 'www_user') <= knows(ID1, A, 'alice') &
5.   hasUser(ID2, 'www_user', 'www', P, R) &
6.   listeningOn(ID3, 'www', 'tcp', '22') &
7.   hostACL(ID4, K, 'www', 'tcp', '22') & hasAccount(ID5, A, K, V)
8. hasAccount(5, A, 'www', 'www_root') <=
9.   hasUser(ID1, 'www_root', 'www', P, R) &
10.  hasAccount(ID2, A, 'www', 'www_user')
11. knows(6, A, 'alice') <= listeningOn(ID1, 'www', 'tcp', '80') &
12.   hostACL(ID2, K, 'www', 'tcp', '80') & hasAccount(ID3, A, K, V)
13. + hasUser(7, 'www_user', 'www', '9JmDGEr4', 'user')
14. + listeningOn(8, 'www', 'tcp', '80')
15. knows(9, A, 'venerus') <= hasUser(ID1, 'www_http_user', 'www', P, R) &
15.   hasAccount(ID2, A, 'www', 'www_http_user')
17. knows(10, A, 'venerus') <= hasUser(ID1, U, 'www', P, 'admin') &
18.   hasAccount(ID2, A, 'www', U)
19. + hasUser(11, 'www_http_user', 'www', 'None', 'user')

```

Figure 3.19: An excerpt of `datalog.py`.

also includes the networks that were originally only implicitly defined. For instance, outside that connects `firewall` to `provider`.

Another TOSCA workflow of CRACK, called *test*, executes the test for the proof trace of Figure 3.20. The result is partially reported in Figure 3.22. The structure of the test follows the facts of Figure 3.20, but it refers to actual runtime values. For instance, the username and password of `www_user` are dynamically configured by `www_weak_enumerable` and `www_weak_password` to `manager` and `qwerty` (respectively). Finally, notice that the test fails. When this happens, CRACK displays the execution log of the failed script for supporting the scenario debugging process.

We now interpret and fix the error that caused the failure of Figure 3.22. From the FUT identifier, i.e., 4, we find in `datalog.json` that the clause was declared by the node `www_weak_password`. The corresponding runtime consists of a script running a dictionary-based brute force over `ssh` by using Hydra [HK18]. From the log, we discover that the error occurred because the password-based authentication is not enabled on `ssh`. The reason is that, by default, `ssh` is configured only to support key-based authentication. To solve this issue, one can add the property `PasswordAuthentication: "yes"` to the node `www_ssh` (see Figure 3.17). Once the property is added, the test concludes successfully.



```

[...]
New fact: hasUser(7, 'www_user', 'www', '9JmDGEr4', 'user')
/* proof of knows(6, 'eve', 'alice') */
New fact: knows(6, 'eve', 'alice')
/* and so on ... */
New fact: hasAccount(4, 'eve', 'www', 'www_user')
[...]
New fact: hasAccount(5, 'eve', 'www', 'www_root')
[...]
New fact: knows(10, 'eve', 'venerus')
[...]
New fact: knows(21, 'eve', 'DB_confidential')

```

Figure 3.20: An excerpt of `goal1.trace`.

### 3.5.4 Execution

We now describe the scenario execution by simulating the attack of the red team. The steps of the attack are depicted in Figure 3.23. Initially, the red team scans the machine hosting the home page of ACME Corp (a). Running `nmap` shows that the server is open on ports 80 and 22. Then, they run the `nmap` script `http-userdir-enum` [Lyo19] and enumerate two users, i.e., *backup* and *manager* (b). The next step (c) is brute forcing the password of *manager* using `hydra` and the `rockyou` [MH18] wordlist (see Section 3.5.3). Once the red team has the password, they can log in `www` as *manager* (d) and execute the privilege escalation discussed in Example 9 (e).

Finally, they leverage the root privileges to read the Wordpress configuration file `wp-config.php` and obtain the administrator access credentials to the database (f). In this way, the red team can browse all the existing records (g) and exfiltrate the data.

### 3.5.5 Evaluation

Below we answer the research questions of Section 1.2.2.

**Answer to RQ1** Our SDL is a steppingstone for the scenario development process. Being a TOSCA extension, CRACK SDL complies with the existing OASIS standard. Thus, IaC developers take advantage of their expertise when designing a scenario and only limited extra skills are required, e.g., for the security-specific aspects. We introduce a design paradigm that can be called *Scenario as Code* (SaC) that extends the IaC paradigm with concepts that are specific to the definition of Cyber Range scenarios, namely vulnerabilities, goals and principals (i.e., teams).

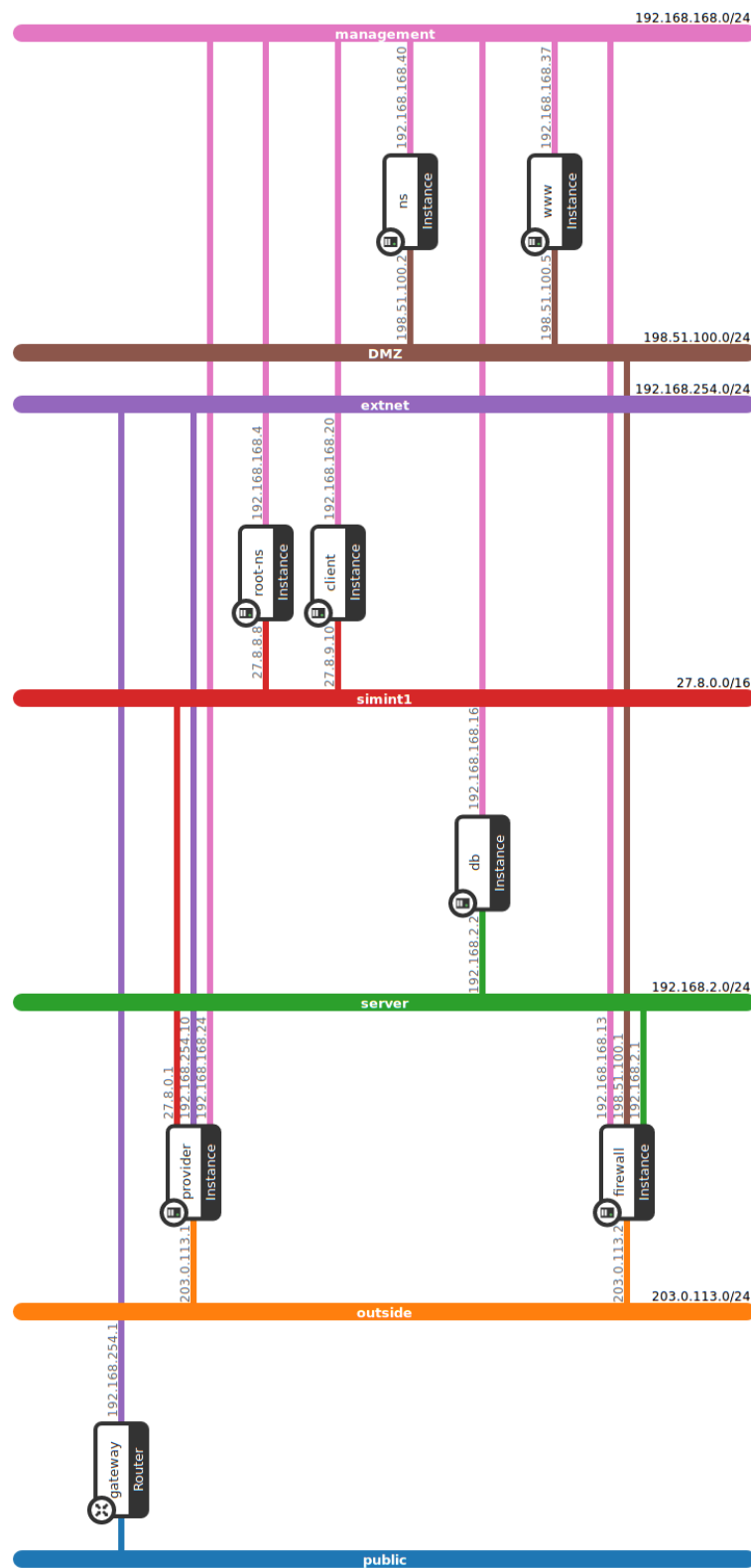


Figure 3.21: The network topology view of the working example theater in OpenStack.

```

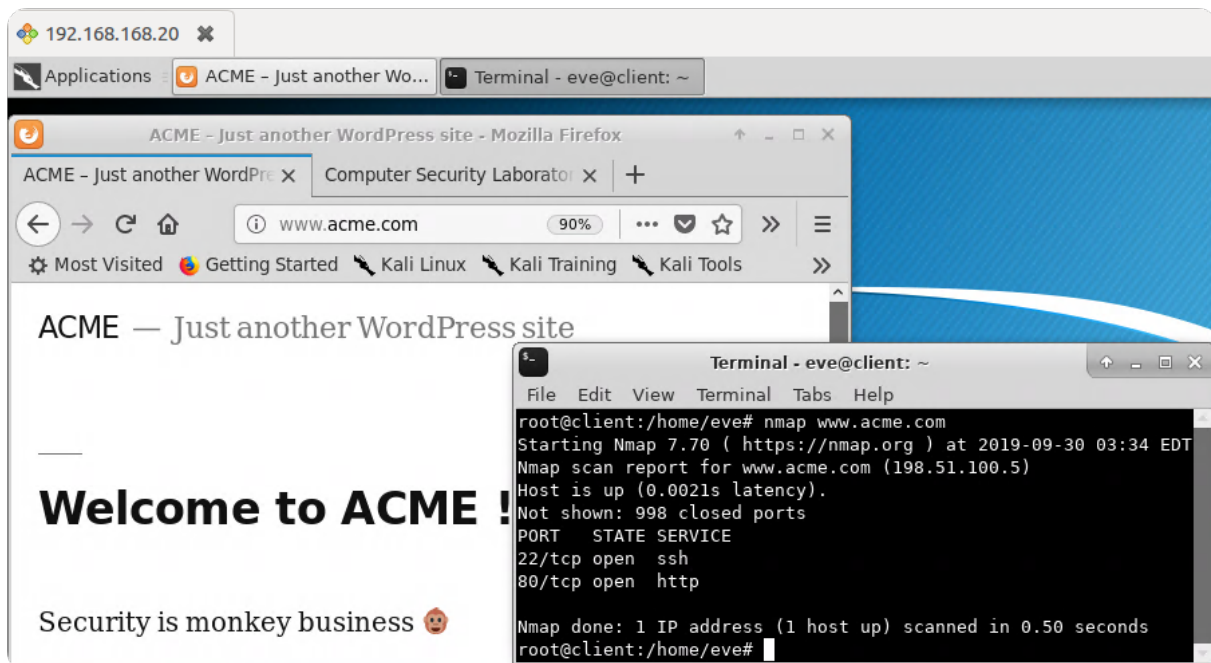
[...]
Verify: hasUser(7, 'www_user', 'www', '9JmDGEr4', 'user')
Execution of test 'hasUser' on www
with parameters 'manager' 'www' 'qwerty' 'user'
Result: OK
[...]
Verify: knows(6, 'eve', 'alice')
Execution of test 'knows' on www
with parameters 'eve' 'alice'
Result: OK
[...]
Verify: hasAccount(4, 'eve', 'www', 'www_user')
Execution of test 'hasAccount' on www
with parameters 'eve' 'www' 'manager'
Result: FAILED
Log:
Hydra v8.1 (c) 2014 by van Hauser/THC
[...]
[DATA] attacking service ssh on port 22
[ERROR] target ssh://127.0.0.1/ does not support password authentication.
[...]

```

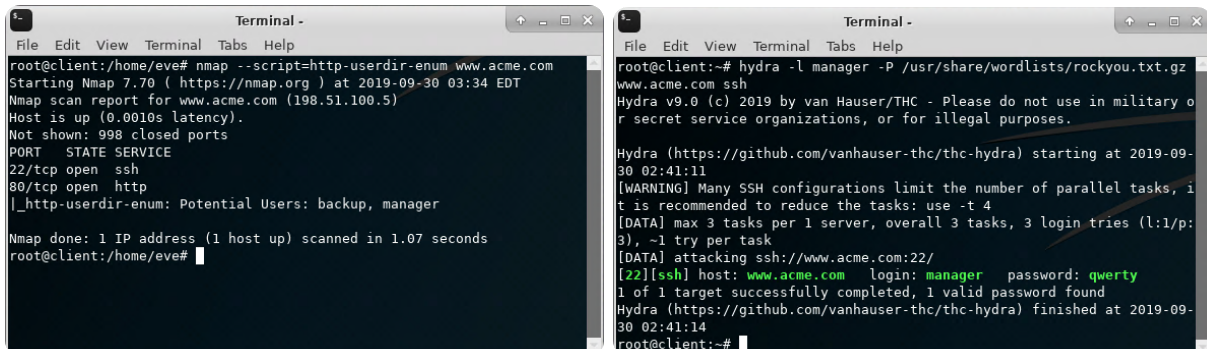
Figure 3.22: An excerpt of `goal1-test.log` containing a failed test step.

**Answer to RQ2** CRACK provides an integrated framework for modeling, verifying, deploying and testing a scenario. On the one hand, our SDL relies on TOSCA for modeling and deploying any infrastructure of interest. On the other hand, SDL extends TOSCA to also model exercise-specific aspects, e.g., vulnerabilities, and verify their interplay. However, since we assume no formal relationship to hold between a model and what is actually deployed, verification is not enough to ensure the desired properties at runtime. Therefore, to effectively support the development, a testing phase is also needed. For this reason, CRACK tests the deployed scenario against the traces generated by the verification process. Although we implemented this approach for Datalog, we stress that other verification techniques are also compatible. Since SDL is extensible, its elements can be labeled with other specification languages. These labels can be composed according to the blueprint topology in order to build global specifications to be verified. As far as the considered verification technique generates evidence, e.g., proof traces, that admit instantiation to an actual test, they contribute to the CRACK development process.

**Answer to RQ3** To answer this question, we start by considering some statistics about our case study. In Table 3.2, for all the scenarios we report (i) the size (i.e., the total number of loc including YAML code and scripts) and the number of types of the scenario blueprint, (ii) the size

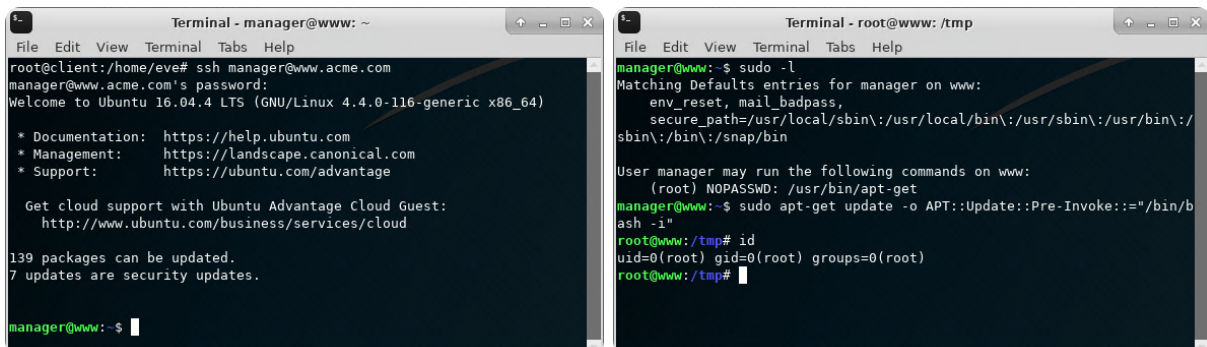


(a) Connecting to client (via remote desktop) and scanning www.



(b) Enumerating the users.

(c) Brute forcing the password.



(d) Accessing www via ssh.

(e) Executing the privilege escalation.

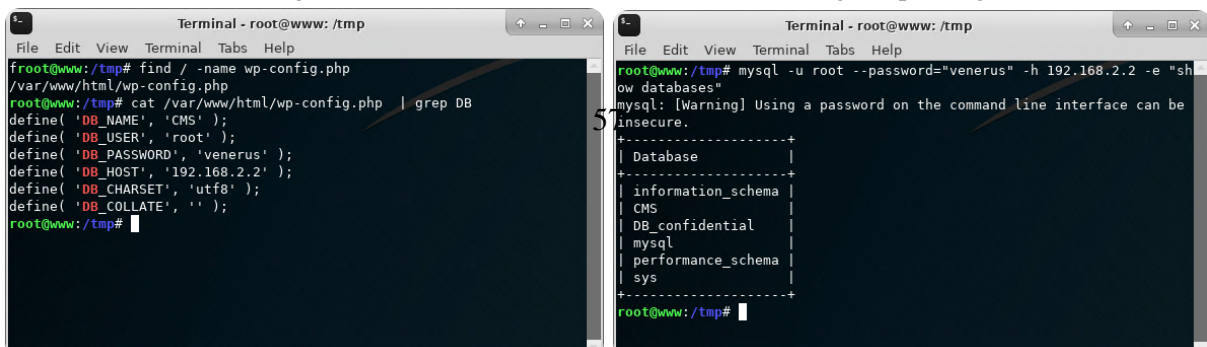


Table 3.2: Numbers of the four scenarios.

#	BLUEPRINT		SPECIFICATION		TEST EXECUTION	
	size	types	size	time	FUTs	time
1	4537	32	275	4.94	205	288.65
2	3764	28	271	4.26	168	224.93
3	4196	28	271	4.12	168	258.67
4	4397	30	270	4.18	215	283.25

Table 3.3: Reuse rate of Scenarios 1, 2 and 3 (w.r.t. Scenario 1).

	SCENARIO 2	SCENARIO 3	SCENARIO 4
loc	3340 (88.73%)	3895 (92.83%)	3924 (89.24%)
types	25 (89.29%)	26 (92.86%)	27 (90.00%)

(Datalog loc) and time<sup>11</sup> (seconds) for the Datalog specification verification, and (iii) the size (number of FUT) and time (seconds) for the test execution. Instead, in Table 3.3 we highlight the reuse rates, both in terms of loc and types, for obtaining Scenarios 2, 3 and 4 from Scenario 1.

The numbers of Table 3.3 highlight that the blueprint of Scenario 1 was largely reused. Intuitively, this happens since a significant portion of all the scenarios deal with a single, shared theater. The same applies to some scenario elements, e.g., the target of the data exfiltration. We claim that, in general, shared elements are frequent, e.g., for scenarios played in a single, specific infrastructure such as ACME Corp. Under these assumptions, by leveraging the type system and the expressive power of our SDL, a designer can reuse the existing blueprints. When reusing existing elements in a different context, our verification and testing procedures ensure that the objectives of the exercise are not compromised.

<sup>11</sup>Specification and test times include both the generation and the execution of the verification/test engine.

# Chapter 4

## Cyber Ranges on the Field

In the previous chapter, we have emphasized the relevance of scenarios in a Cyber Range. In particular, Next Generation Cyber Ranges can offer facilities for assisting and simplifying the generation of complex scenarios making them ever closer to reality and easily modifiable. The above properties represent the ideal conditions for creating sandboxes, i.e., *testbeds*, where components (e.g., part of real infrastructures, devices, or configurations) can be tested against security threats. Testbeds are mandatory when the components under evaluation are part of real operational critical systems.

An emerging paradigm that typically interacts with critical infrastructure components is Fog Computing. Testing in-depth the possible security weaknesses of a Fog device means to create a testbed that faithfully reproduces the typical operating conditions, including critical components.

In this chapter, we leverage the capabilities of Cyber Ranges and hosted scenarios to analyze the security threats of a Fog-based solution. In Section 4.2, we extend the case study with components required for creating a testbed which emulates a smart agriculture system, one of the vertical markets to which Fog computing is expected to be fruitfully applicable [Bye17]. This extension allows us to examine the security model of a mainstream Fog operating systems and point out some weaknesses as detailed in Section 4.3. Then, in Section 4.4, we proposed a technique to reduce the identified attack surface. The same testbed allows us to perform an experimental evaluation and verify the effectiveness of our solution (see Section 4.4.3).

### 4.1 Practical Fog Computing

In this section, we provide some background on Fog Computing (Section 4.1.1), Fog applications (Section 4.1.2), and Fog operating systems (Section 4.1.3) with a specific focus on Cisco IOx.

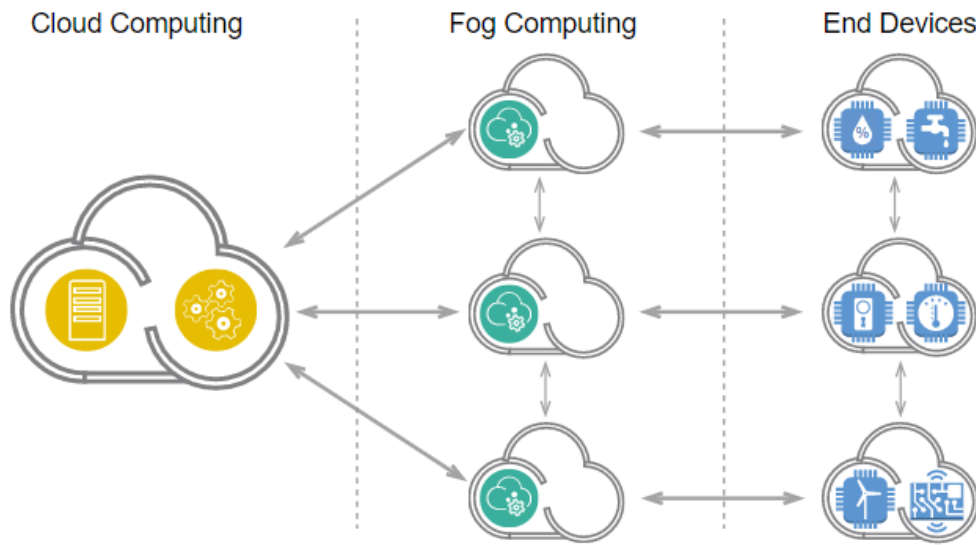


Figure 4.1: The Fog Computing paradigm.

### 4.1.1 Fog Computing

The emerging Internet of Things (IoT) paradigm [SPB19] is changing the way data are produced and analyzed, and the way application are designed, implemented and deployed. It is transforming centralized, monolithic server applications managing complex IT/OT environments into a set of smaller, device-specific, interacting applications. Each IoT application runs on its own hardware and operating system, manages its own dataset and continuously interact with the other apps.

Unfortunately, the adoption of the Cloud Computing paradigm in this setting is sub-optimal as it requires moving data produced by IoT applications from the edge to remote data centers (i.e., in the Cloud) and vice versa, in order to be analyzed and take decisions. This approach introduces delays that affect interactions between IoT applications and may lead to the failure of jobs with specific time constraints (e.g., soft real-time).

*Fog Computing* overcomes this problem by introducing an intermediate computing level (see Figure 4.1) provided by specific devices, i.e., *Fog nodes*, that provide computational power to analyze data just where they have been produced. In this way, delays are greatly reduced while the advantages of decentralized computation are retained.

Fog Computing addresses the need of advanced IoT ecosystems where applications can produce large streams of data that must be analyzed quickly. More specifically, each Fog node deals with a subset of an IoT ecosystem, namely a set of IoT devices belonging to the same local network, and context-aware applications working on data produced by the subset of IoT devices.

From a technical standpoint, Fog Computing is decentralized, heterogeneous, and often domain-specific, thereby resulting very different from the centralized approach of Cloud Computing. In a nutshell, each Fog node has a multi-programmed OS that allows executing several applications concurrently. A de-facto standard Fog OS at the time of writing is *Cisco IOx* [Cis19a].

In the following, we detail Fog applications (see Section 4.1.2) and Cisco IOx (see (Section 4.1.3).

### 4.1.2 Fog Applications

Fog nodes allow for executing applications in different environments which also depend on the manufacturer and the characteristics of the hosting devices. As described in [RVM19], Fog devices typically support the execution of at least the following packaged applications:

- *Virtual Machine (VM) packaged applications*, that consist of a traditional virtual machine containing an operating system, libraries and application code. Fog devices can host a hypervisor to run such packaged applications.
- *Platform as a Service (PaaS) style applications*, which are self-contained programs developed using high-level languages, e.g., Java, Python, and Ruby. Fog devices provide them with the execution environment as a service.
- *Container applications*, that depend and are designed to execute directly on the operating system of the Fog node. This solution, as opposed to PaaS style, depends on the features of the Fog operating system, but it leaves the complete flexibility to developers on the choice of the programming language, as well as the full framework stack. Fog devices often leverage the Linux Container (LXC) paradigm [Ber14].

Each application, regardless of the packaging method, also embeds a standard descriptor file containing the hardware requirements that the application needs from the Fog node, in order to execute properly (e.g., computing and network resources).

Beyond some specific features related to the execution environment, a Fog application is made of a set of standard components. Figure 4.2 depicts such components as well as their interaction with the Fog environment.

In particular, one main component, namely the *Field Device Connector*, interacts with edge devices and acquire raw data. Moreover, further data can be retrieved directly from the Fog platform itself, e.g., from services like GPS or other connected serial devices, using a *Runtime Support* component.

A *Data Processing* component is in charge to elaborate the above inputs in order to apply configurable rules of filtering, reduction, and analysis. The output is then processed by a *Data*



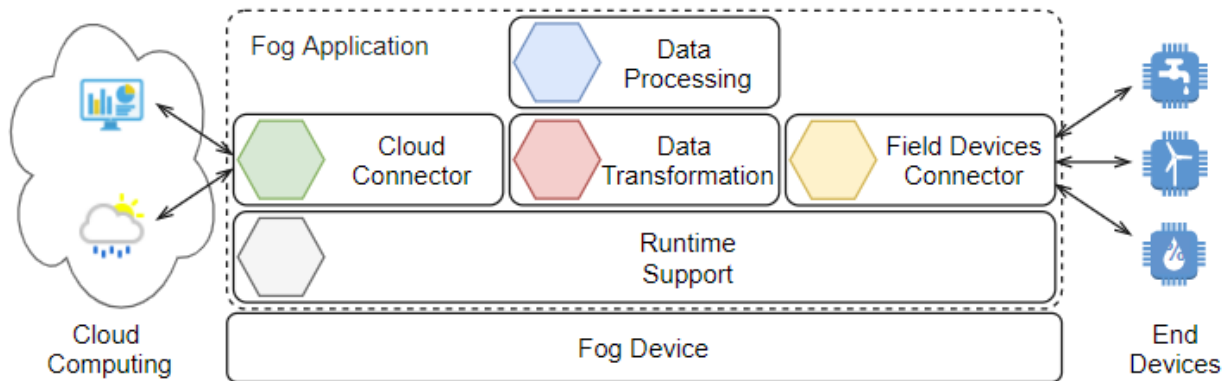


Figure 4.2: Common components of a Fog application.

*Transformation* component which applies some custom business logic to render acquired data using a standard representation format, like, e.g., JSON or XML. This transformation allows a *Cloud Connector* to send the processed data to a centralized application, hosted in the Cloud or in an external data-center, using a web-friendly protocol, like HTTP. At the same time, the Cloud Connector provides a web interface, e.g., a RESTful web service, which allows the Cloud or an external data-center to interact with the Fog application itself.

Notice that the modular approach of Fog applications follows the *microservices* approach [NSS14], which is likewise adopted in the IoT environment [JKAC18].

### 4.1.3 Cisco IOx

Cisco IOx [Cis19a] combines the traditional Cisco Internetwork Operating System (IOS) software [Cis19c] with the Linux Operating System in a single network device, e.g., routers or switches. The main objective of such a solution is enabling the hosting of applications on devices running at the network edge.

Figure 4.3 depicts the components of a device running Cisco IOx [Cis18]. The Cisco IOS component provides functionalities for network services, e.g., connectivity with the physical network and routing protocols, and core security services, e.g., Network Address Translation (NAT) and a firewall. The main task of the Linux component is managing the resources, e.g., computing and storage, and providing a framework for executing Fog applications. To this aim, a specific layer, namely *Application Management*, covers all life cycle aspects of applications including development, distribution, deployment, hosting, monitoring, and management. Moreover, a middleware, namely *Cisco Application Framework* (CAF), exposes an interface for accessing the functionalities of the above layer. A command-line client (*ioxclient*) and a local web interface (*Local Manager*) use the CAF to allow authenticated users to manage the IOx device.

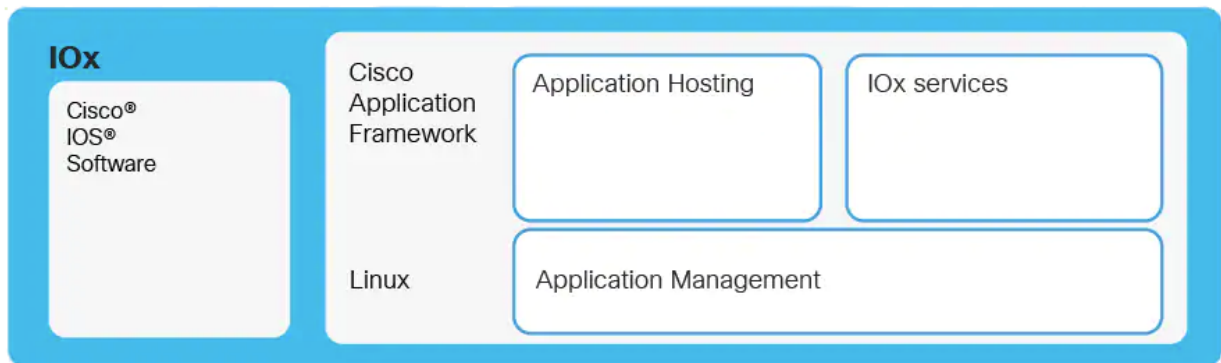


Figure 4.3: Cisco IOx architecture (as given in Cisco IOx Data Sheet [Cis18]).

Finally, an IOx node supports the execution of custom Fog applications and internal services, namely *IOx services* [Cis19a], in a PaaS-style or LXC-style deployment. IOx services extend the Cisco IOx operating system and create the basic infrastructure for a microservices ecosystem. In particular, these services provide an implementation of the common components of a Fog application (see Figure 4.2) and enable their interaction through a specific module named *North Bound Interface* (NBI).

#### 4.1.4 Cisco NBI

The Fog model relies on applications and services, running in different containers, interacting with one another and with the field devices. For this reason, each Fog implementation usually needs to execute a *Message Broker*, i.e., a shared and authenticated communication channel for enabling such interaction among the involved entities.

Cisco IOx implements the Message Broker through the NBI gateway provided by the IOx services. In detail, the NBI service allows publish and subscribe semantics, thereby using *Message Queue Telemetry Transport* (MQTT) [PPR<sup>+</sup>19].

MQTT is a lightweight messaging protocol that adopts the publish-subscribe pattern: messages sent by publishing clients to volatile messaging queues—called topics—are received and (possibly) routed by a broker to topics-subscribed clients. Topics, specified as UTF8-type strings, are hierarchically organized into levels (using the forward slash) and are used by the broker to filter and organize incoming messages.

Therefore, the NBI allows clients to execute four actions: *create* a topic, *delete* a topic, *publish* a message, and *subscribe* to a topic. To publish messages, Cisco uses JSON [Bra17] as the default notation. Accordingly, each component interacting with the NBI provides a *data model* [Cis19a] for describing the content format of its own JSON messages. However, notice that data mod-

els represent only an agreement, and each application or service is responsible for publishing messages according to their own model.

We refer the reader to Section 4.2.2 for an example of data models and exchanged messages.

## 4.2 Motivating Scenario

In this section, we introduce a motivating scenario concerning the field of smart agriculture. Therefore, we provide an implementation based on the Fog paradigm.

### 4.2.1 A smart agriculture system

Our motivating example, inspired by [KST<sup>+</sup>18], is a lifelike scenario in which sensors, data acquisition and analytics contribute to the implementation of a smart agriculture system.

In this scenario, *end devices* belong to two categories. *Sensors* measure the moisture status of the soil, while the *irrigation system* is used for the watering of the soil. A *smart agent*, implemented by a software application, acquires data from the sensors and, if the soil moisture level falls below a certain threshold, it activates the irrigation system. The smart agent also interacts with remote data sources providing the *weather forecasts*. This allows the agent to plan the duration of watering and optimize water consumption. Finally, the smart agent delivers data, e.g., the number of irrigation actions per day or the average water consumption, to a remote cloud application that provides the *data visualization* service.

### 4.2.2 Fog implementation

The functionality provided by a standard Fog deployment fits the requirements of our motivating scenario: in fact, the smart agriculture system needs to gather and analyze few bunches of data from local and isolated sensors (i.e., moisture) seamlessly, and requires limited computational resources to make a decision.

To this aim, we extended the case study introduced in Section 3.1 with the components required for deploying the smart agent as a Fog application. The resulting infrastructure is depicted in Figure 4.4.

A physical *Fog device* is connected to the Server network. It provides a new *virtual* network, namely *App*, used by the running Fog core services and applications. The IoT network hosts the end devices used in the smart system, i.e., a moisture *sensor* and an *irrigator*. A weather

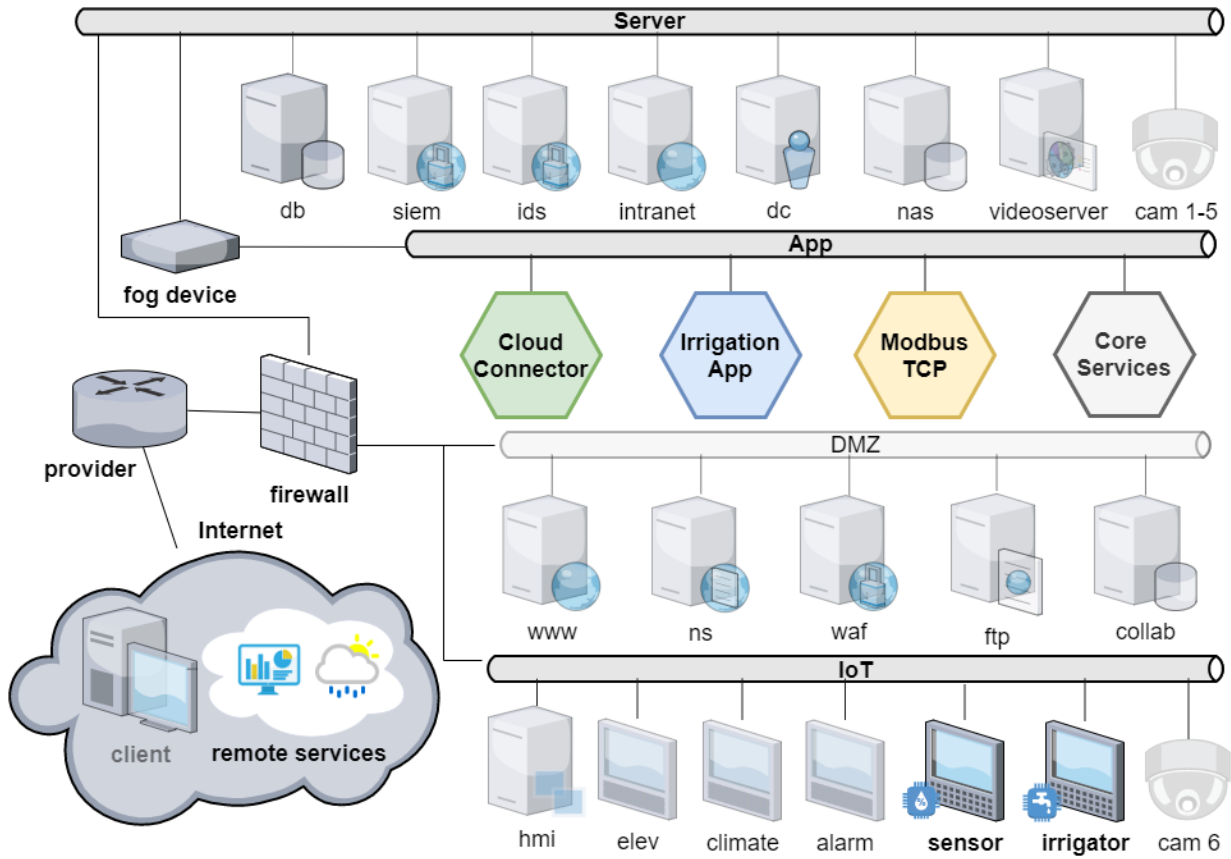


Figure 4.4: Fog implementation of the smart irrigation infrastructure.

forecast and a data visualization services, i.e., *remote services*, are reachable through the Internet connection. Below we detail each new component.

**Fog device.** This device represents the core element of our implementation. It provides the connectivity at the network edge and the capabilities for running Fog applications. We employed a Cisco IR829GW-LTE-GA-EK9 [Cis19b] running IOx software release 15.9(3)M, which is an industrial router having connectivity for different technologies, e.g., Ethernet, serial, and wireless, together with the computing capabilities for running PaaS, e.g. Python, and Java, and container-based, e.g., Docker, and LXC applications. These capabilities are provided through a Linux-based and embedded IOx [Cis19a] operating system version 1.9.0.5.

This device can reach remote services through the Internet connectivity provided by the corporate firewall. It interacts at the same time with the end devices using the Modbus communications over TCP (Modbus TCP) [S<sup>+</sup>99] protocol. Moreover, it achieves segmentation between the

connected zones using the following firewall rules configured on the IOS component.

1. Only the Remote Data Sender/Receive application can contact the public Internet.
2. Only the Modbus TCP application can contact the IoT network.
3. App network is isolated, i.e., the running applications cannot be contacted from the other connected networks

**End devices.** The soil sensor measures the moisture level, and the irrigator executes the watering. We simulate these devices as two slaves Modbus Programmable Logic Controllers (PLC). In particular, the sensor has one read-only register containing the simulated moisture level, while the irrigator has two writable registers for activating watering and setting the duration, respectively.

**Modbus TCP and Cloud connector.** They represent two applications designed as a general microservice for establishing Modbus TCP and executing remote Application Program Interface (API) calls through the HTTP protocol, respectively. Each service supports the configuration of multiple data sources, e.g., a Modbus device or a remote HTTP endpoint. For each data source, the service requires its address, e.g., an IP address or an HTTP Uniform Resource Locator (URL), and the data model.

```
1. {"dataSchemaId":"MoistureSchema",  
2.  "description":"Data Schema for sensor emitting moisture values",  
3.  "fields":[{"name":"moisture",  
4.    "type":"short"}]}
```

Figure 4.5: Data schema of a moisture sensor.

For instance, Figure 4.5 represents the data model, namely `MoistureSchema`, for a generic moisture sensor. In particular, it emits the moisture values using a field named `moisture` (line 3) containing a short integer (line 4).

Moreover, Figure 4.6 represents the data model of a Modbus (line 2) PLC device, named `Moisture_PLC` (line 1), with the capability of measuring the soil moisture. It contains a single sensor (lines 3-13), named `MoistureSensor` (line 4), using the data model presented in Figure 4.5. In particular, the moisture value (line 9) is emitted reading the analog register addressed by the range contained in the `registerRange` field (lines 11-13).

```

1. {"deviceId": "Moisture_PLC",
2.  "protocol": "modbus",
3.  "sensors": [{
4.    "name": "MoistureSensor",
5.    "dataSchemaId": "MoistureSchema",
6.    "contentHandler": {
7.      "contentType": "raw",
8.      "contentMappings": [{
9.        "fieldName": "moisture",
10.       "protocolProperties": {
11.         "style": "analog",
12.         "mode": "read",
13.         "registerRange": "40001-1"}}]]]]}

```

Figure 4.6: Data schema of a PLC containing a moisture sensor.

**Core services.** This container executes IOx services (see Section 4.1.3) and the NBI service (see Section 4.1.4).

```

1. {"topic": "Moisture",
2.  "context": {
3.    "deviceId": "PLC1",
4.    "sourceId": "MoistureSensor",
5.    "dataSchemaId": "MoistureSchema",
6.    "timestamp": "1577730537"},
7.  "message": [{ "values": [10]}]}

```

Figure 4.7: A NBI message related to the soil sensor.

As an example of interaction with the NBI service, Figure 4.7 represents a JSON message published in the `Moisture` topic (line 1) by the Modbus TCP application. It is related to the moisture level read from the PLC measuring the soil moisture, named `PLC1` (line 3). In particular, the value is read from the `MoistureSensor` (see Figure 4.6) of the PLC (line 4) and it is formatted (line 5) according to the `MoistureSchema` (see Figure 4.5). The `timestamp` (line 6) identifies the moment in which the read event occurred and the message field contains the level value (line 7) expressed as a short integer.

**Remote services.** They serve as external platforms used for visualizing data and acquiring weather forecasts. We simulate the data visualizer using the *ThingSpeak*<sup>1</sup> public Cloud service, which enables the storing and visualization of data using remote HTTP API calls.

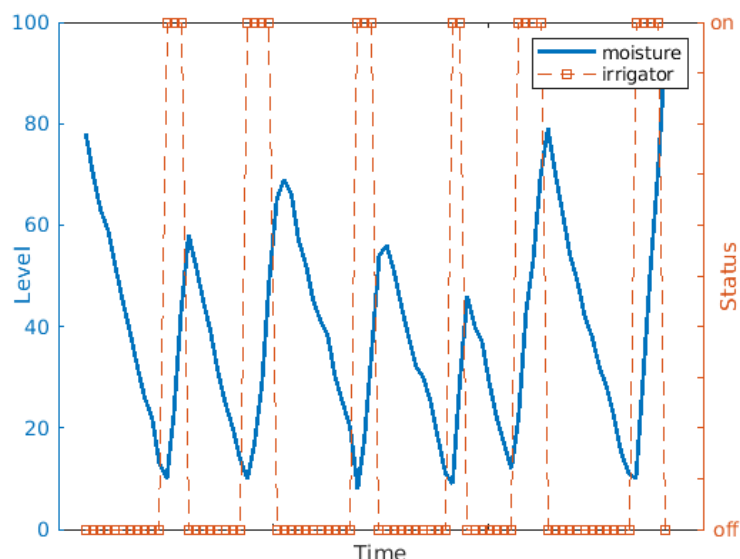


Figure 4.8: Data visualization for the moisture value and the irrigator status.

Figure 4.8 represents the data visualization<sup>2</sup> for the moisture value and for the status of the irrigator. Notice that every time the Irrigation App acquires a low value of the simulated moisture, it activates the irrigation.

**Irrigation App.** It implements the functionalities described in Section 4.2.1 for the smart agent. This application relies on the following topics created on the NBI.

- *Moisture*. The Modbus TCP reads moisture values from the sensor and write a corresponding message on this topic.
- *Irrigator*. The Modbus TCP uses this topic to send data to the irrigator.
- *Wxf*. The Remote data sender/receiver reads weather forecasts and publishes a corresponding message on this topic.
- *Visualizer*. The Remote data sender/receiver consumes this topic for sending data to the ThingSpeak service.

<sup>1</sup><https://thingspeak.com>

<sup>2</sup>In Figure 4.8 we give no time scale since we used a simulated time in our experiments.

Briefly, it executes this endless loop: *i*) read messages from the moisture topic and extract its level, *ii*) if the level is low, then read weather forecasts from WxF topic for calculating duration, and publish to the irrigator topic the command for starting watering, *iii*) write a message to the visualizer topic containing aggregate data about the moisture level and irrigation time.

## 4.3 Security Model

Fog applications enable the possibility to implement the business logic to monitor and control the on-field devices, and to interact with cloud appliances.

In our motivating scenario, the core logic of the smart irrigation infrastructure resides in the Irrigation App, which can monitor the IoT sensors, read the acquired data and instruct the actuators (i.e., the irrigation system) accordingly. Furthermore, the Remote Data Sender application is in charge to communicate with the remote cloud services and deliver data concerning the weather forecast to the Irrigation App.

Indeed, from a security standpoint, the behavior of the Fog applications and in particular their interaction with *i*) the core services, *ii*) the field devices, and *iii*) other applications (either remote or local) represent a critical attack vector for the whole infrastructure. Furthermore, Fog applications are usually provided in a black box fashion by third party suppliers or adapted from legacy software, and their installation process is straightforward, thereby resulting in a powerful attack vector.

### 4.3.1 NBI Security Features

The IOx Core Packages provides NBI module as the main conduit for enabling secure communication between applications and services. In details, NBI is a security mechanism responsible for *i*) authenticating applications, and *ii*) authorizing message delivery and receipt. From a general point of view, NBI allows the enforcement of a *global security policy* over the communications among Fog applications and services, as stated in [Cis19a].

Applications and services can interact with NBI through REST and websockets [MF11] APIs that are authenticated using the OAuth protocol following the OAuth 2.0 specifications [Har12]. The connections to both the REST and websocket endpoints are secured by enabling TLS to secure the transport. At installation time, for each application declaring the dependency in its package.yaml, NBI assigns *i*) individual OAuth credentials, and *ii*) NBI IP address and port.

An OAuth access token is required to make successful calls to NBI. Applications use their credential to obtain a Bearer Access Token and use it to send publish/subscribe requests to NBI IP address and port. NBI accepts incoming requests only after validating the access token.



### 4.3.2 Weaknesses of NBI

Since NBI acts as the main gateway for the intercommunication between applications and services of the Fog node, we focused our investigation on the security mechanisms enforced by NBI.

Our analysis confirms that any Fog application can access NBI only by providing a valid OAuth access token. As stated in [Cis19a], IOx only supports Bearer access tokens, but the lifetime of the access token is indefinite.

As described in [YM13], Bearer tokens are considered less secure than Message Authentication Code (MAC) tokens [J. 12] as anyone with the token can have access to the protected resources, without being further authenticated. To protect against token spoofing and stealing, the specification requires that the TLS mechanism must be adopted when transmitting the access token and the protected resource requests [JH12].

Unfortunately, although the connections to both the REST and websocket endpoints support TLS, even in the most recent IOx version, the supported certificates are self-signed, thus exposing the communication to Man-in-the-Middle attacks (*T1 - Man in the Middle*) [The19c] and consequently to the stealing of the token. Furthermore, as the lifetime of the access token is unbounded, the attack surface increases, since it enables an unlimited and unrestricted usage of stolen tokens.

Regarding the security of the publish/subscribe model adopted by NBI, our findings suggest that NBI does not enforce any restriction on the creation and subscription to topics, thus allowing any application to potentially subscribe to topics created by any other application or service of the Fog Node. This behaviour is also suggested by the official Cisco documentation that describes a way to subscribe to all sensors of the Fog node [Cis19a].

To prove that, we developed and installed a malicious Fog application (i.e., *evilapp*) implementing some of the attacks discussed in [ARH17]. In details, the malicious application was able to sniff all the data (*T2 - Sniffing Attack*) [The19b] transmitted on all the available topics, by exploiting the wildcard character. In addition to this, we also discovered that our malicious application was able to publish fake data (*T3 - Data Manipulation*) on any existing topic. For instance, the above activity allows the attacker to inject false information, (*T3.1 - Data Injection*), cause unauthorised alteration of data (*T3.2 - Data Tampering*), re-transmit legitimate packets (*T3.3 - Replay Attack*), send unexpected verbs or input values (*T3.4 - Inject Unexpected Items*) to the involved components.

Finally, for the same reason, a malicious application can publish fake messages assuming the identity of some other components of the system (*T4 - Identity Spoofing*) [The19a].

## 4.4 Methodology

As described in the previous section, the actual implementation of the NBI service suffers from several weaknesses. Some of them can be solved by adopting proper state-of-the-art solutions for managing tokens expiration, MAC tokens, and trusted certificates.

In this respect, the scientific community has proposed several solutions for the MQTT protocol *i)* to improve end-to-end encryption (e.g., [SCC19] and [MNE<sup>+</sup>17]), *ii)* to enhance authentication and authorization mechanisms (e.g., [NIP<sup>+</sup>16] and [CPVV18]), or *iii)* to introduce access control mechanisms (e.g., [CF18] and [LMMM<sup>+</sup>17]).

Furthermore, some implementations of the MQTT standard can improve the reliability of the publish/subscribe paradigm. For instance, Mosquitto [Ecl19] and HiveMQ [Hiv19] implementations support the verification of SSL/TLS certificates as well as some basic authorization mechanisms.

Unfortunately, previous solutions still represent an unsatisfactory remediation, since the Fog environment requires a *message broker supporting stateful and fine-grained policies*, e.g., based on message content or depending on the current state of the Fog ecosystem. Furthermore, such a broker has to be well-integrated with the other existing components, e.g., the implemented authentication layer.

In this section, we present an extension of the standard NBI and an embedded run-time enforcement methodology called DIOXIN (Discerning IOx INterface), that is able to supervise the interaction among Fog application components according to a set of security rules.

### 4.4.1 Overview of the approach

Figure 4.9 depicts the abstract workflow of DIOXIN. The workflow activates once the standard NBI receives a message, and it is executed before carrying out the requested action.

In the first task, a component, namely the *NBI Message Adapter* (MA), encapsulates the original message according to an ad-hoc format named *NBI Message Schema*. Briefly, MA extends each message with a set of metadata collected from the request made by the sender (e.g., the identity of the sender and the details on the topic). The extended *NBI message* is then delivered to the *NBI Policy Decision Point* (PDP). In details, PDP evaluates the incoming request according to a set of security policies that depends on *i)* the specific features of the message, i.e., *scopes*, *ii)* the overall state of the running Fog ecosystem, i.e., the *context*, and *iii)* the content of the NBI message. This task completes by returning an authorization decision, i.e. deny, allow, or log, related to the requested action.

Finally, the authorization decision is enforced by the *NBI Policy Enforcement Point* (PEP), which

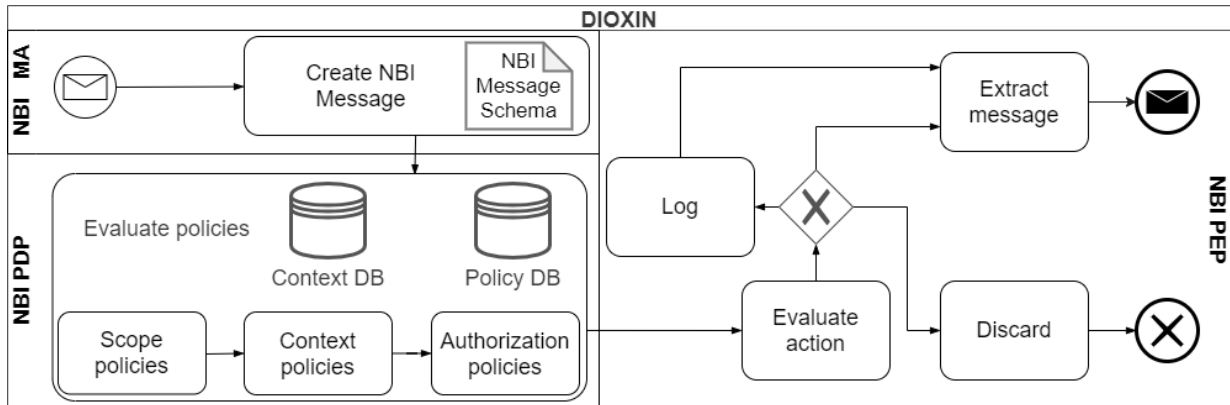


Figure 4.9: The workflow of DIOXIN.

*denies* (i.e., the message is discarded), *allows* (i.e., the original message is sent to the standard NBI), or *logs* (i.e., equivalent to 'allow', but it also generates a log message) the incoming message.

## NBI MA

```

1. {"$id": "https://csec.it/schemas/nbimessage.schema.json",
2.  "$schema": "http://json-schema.org/draft-08/schema#",
3.  "title": "NBI Message",
4.  "type": "object",
5.  "properties": {
6.    "topic": {"type": "string"},
7.    "senderId": {"type": "array"},
8.    "action": {"type": "string", "enum": ["create", "delete", "
↪ publish", "subscribe"],
9.    "timestamp": {"type": "string"},
10.   "message": { "type": "object"}}}

```

Figure 4.10: The JSON schema of an NBI message.

NBI MA creates a new JSON message according to the NBI Message Schema, represented in Figure 4.10.

The schema, named *NBI Message* (line 3), has an unique identifier (line 1) and follows the draft-08 version (line 2) of the JSON Schema.

In detail, a message contains a JSON object (line 4) with the following properties (line 5-13).

- *Message topic.* The `topic` property (line 6) must contain a string with the identifier of the topic of a message.
- *Sender identifier.* The `senderId` property (line 7) must contain an array of strings with the unique identifier and the aliases of the sender.
- *Timestamp.* The `timestamp` property (line 8) refers to the creation time of the NBI message.
- *Action.* The `action` property (line 8) must contain the client requested action, i.e., `publish a message` or `subscribe to a topic`.
- *Message content.* The `message` property (line 10) contains an object related to the original JSON message. Creating and deleting a topic does not apply to messages and, in such instances, this property can be empty. Moreover, notice that any property `prop` of the original message can be still accessed through the path `message.prop`<sup>3</sup>.

We stress that DIOXIN sets the properties of an NBI Message only after the original request is delivered to NBI, thus minimizing the risk of being tampered by an external entity (see T3 in Section 4.3.2).

**Example 11.** Consider the malicious application *evilapp*, introduced in 4.3.2, installed and running on the Fog node. This application can exploit T2 (see Section 4.3.2) and sniff messages related to the soil sensor (see Figure 4.7). Then, the application publishes the sniffed message with a manipulated moisture value and exploits T3 (see Section 4.3.2).

Figure 4.11 shows the message published by *evilapp* after it is extended by the MA. In particular, the original message is copied in the `message` property (line 4-11) and the `senderId` property (line 2) contains the `client_id` and the application name of the *evilapp*. Notice that, instead of the original message, it is now clear that an external application is trying to interact with the irrigation system. □

## Message PDP

The Message PDP evaluates a list of enforceable policy rules and returns an authorization decision against an NBI Message.

The general form of a policy rule is defined as follows.

---

<sup>3</sup>We use the JavaScript dot notation for accessing JSON objects

```

1. {"topic": "Moisture",
2.  "senderId": ["dbd8b209-9970-4a5e-9527-ea8e6097fd50", "evilapp"],
3.  "action": "publish",
4.  "message": {
5.    "topic": "Moisture",
6.    "context": {
7.      "dataSchemaId": "MoistureSchema",
8.      "deviceId": "PLC1",
9.      "sourceId": "MoistureSensor",
10.     "timestamp": "1577730537"},
11.    "message": [{"values": [90]}]}
```

Figure 4.11: A message extended according to the NBI Message Schema.

```
<result> IF (<test>)
```

The main task of the PDP is evaluating the `test` predicate and, if it is true, returning the corresponding `result`. In particular, a result can be a list of predefined expressions which amount to *i*) adding a scope to the message (*scope policy rule*), *ii*) updating a value in the context (*context policy rule*), and *iii*) returning an authorization decision (*authorization policy rule*).

Similarly, a `test` predicate can be expressed according to *i*) the scopes of the NBI message, *ii*) the values in the context, and *iii*) a property value in the NBI message. Below we give a detail of each class of policy rule.

**Scope policy rule** A *scope* is an identifier used for labelling messages. It represents a logical way to characterise messages according to common properties like, e.g., values from PLCs or sensitive devices.

**Example 12.** We want to label messages related to the field devices of the irrigation system, i.e., the moisture sensor PLC1 and the irrigator IRRIGATOR, with the IRRIGSYS scope.

```
1. SCOPE.push("IRRIGSYS") IF (message.context.deviceId == "PLC1" OR
   message.context.deviceId == "IRRIGATOR")
```

The above scope policy rule adds the IRRIGSYS scope to all the NBI messages containing a property `context.deviceId` with PLC1 or IRRIGATOR as value in the original message, according to the data schema of devices (see Figure 4.6). □

**Context policy rule** The Message PDP keeps a database of global variables, namely the *context*, representing the state of the Fog system. The values of the context variables depend on the content of the incoming NBI messages. To identify a message containing the value of a context variable, we use context policy rules.

**Example 13.** We want to save the value of the moisture in the CTX context array. This allows creating policy rules depending on its actual value.

```
2. save_value(CTX["moisture"], message.message[0].values[0]) IF (
    topic == "Moisture" AND SCOPE.includes("IRRIGSYS") AND message.
    context.dataSchemaId == "MoistureSchema")
```

The above context policy rule saves the value of the moisture, included in the original message, in the `moisture` index of the CTX array if the message *i*) has been published in the `Moisture` topic, *ii*) contains `IRRIGSYS` in its scopes (see Example 12), and *iii*) has a `MoistureSchema` as data schema (see Figure 4.6), i.e., it has been generated by a moisture sensor of the irrigation system. The `save_value` represents an internal function that updates the context if and only if the timestamp of the message is more recent than the one associated with the value of the context variable. □

**Authorization policy rule** The clause of an authorization policy rule determines if a requested action, i.e., publish, subscribe, create or delete a topic, could be authorized (*allow*), denied (*deny*) or authorized and logged (*log*).

The design of such policy rules depends on a global policy, namely the *default policy*, which determines what happens when an input message does not match any other rule. If the default policy is deny, then the authorization policy rules will define allowed actions and vice versa.

**Example 14.** Consider a default deny policy. We want to enforce allowed actions on the `Moisture` topic.

```
3. ALLOW IF (topic == "Moisture" AND senderId == "Modbus TCP")
4. ALLOW IF (topic == "Moisture" AND action == "subscribe" AND
    senderId == "Irrigation App")
5. ALLOW IF (topic == "Moisture" AND SCOPE.includes("IRRIGSYS") AND
    action == "publish" AND senderId == "Irrigation App" AND CTX["
    moisture"] < 30)
```

In the above three policy rules, the first allows any action if the message comes from the Modbus TCP service, i.e., this service can publish values read from sensors and receive messages containing commands for field devices. The second states that the `Irrigation App` can subscribe to the `Moisture` topic and receive messages about the moisture sensor.

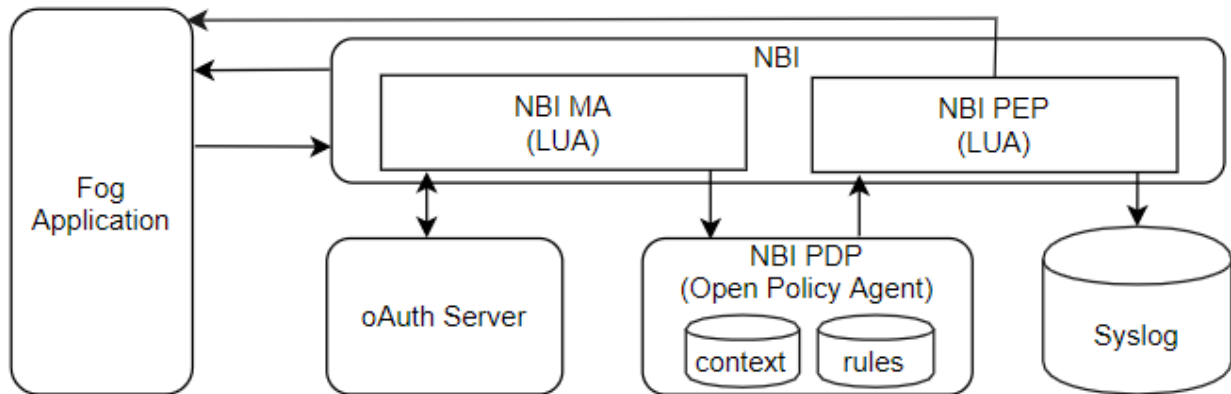


Figure 4.12: The architecture of DIOXIN implemented on Cisco IOx.

The last allows the `Irrigation App` to publish messages on the `Moisture` topic, i.e., start the watering, if and only if the scope of the message is related to the irrigation system and the actual value of the moisture is less than a fixed value.  $\square$

### Message PEP

The NBI PEP receives the NBI Message and the related authorization decision from the NBI PDP. Therefore, it executes a corresponding action. If the authorization decision is `deny`, the NBI PEP discards the message, creates a log entry about this violation, and return an error code to the sender. If the authorization decision is `allow` or `log`, the NBI PEP extracts the original message from the NBI message and sends it to the standard NBI for following the ordinary flow. Also, if the authorization decision is `log`, the NBI PEP creates a log entry related to the message.

**Example 15.** Consider the `evilapp` application described in Example 11 and the rules introduced in Examples 12, 13, and 14. If we enforce a default deny policy, the `evilapp` can not sniff and publish messages because any of the allow rules matches her `senderId`. For this reason, the NBI PEP blocks and logs the `evilapp` attempts to exploit T2 and T3.  $\square$

## 4.4.2 Implementation

We present here a proof-of-concept implementation of DIOXIN on Cisco IOx devices.

Briefly, it includes an extension of the standard NBI which embeds the NBI MA and NBI PEP modules, and a new component running the PDP module (see Figure 4.12). As described in

Section 4.1.3, the NBI service runs inside the IOx services package provided by Cisco. In particular, this package consists of a Linux container hosting a LUA [IDFF96] web application. This application provides the HTTP endpoints for Fog Applications interacting with the NBI and implements the publish/subscribe and create/delete topic actions (see Section 4.1.4). Therefore, we added inside the LUA code a new function being called before the standard request handlers execute the actions required by Fog applications. The above function performs the activities related to the NBI MA and NBI PEP, namely:

1. It receives as input the message sent by an application along with the bearer token used for authentication.
2. It executes the existing authentication procedure with the IOx OAuth server and extracts the `client_id` of the authenticated application from the response.
3. It uses the `client_id` together with the `topic`, the requested `action`, and the `timestamp`, and creates the NBI Message (see Section 4.4.1).
4. It requests the evaluation of the NBI Message by sending an HTTPS request to the NBI PDP.
5. If the PDP returns `deny` as an answer, the PEP sends back a forbidden answer to the Fog Application. Otherwise, it invokes the standard request handlers with the original message. Moreover, if a `log` request is returned, the function also writes a message to the IOx Syslog.

The PDP component is implemented through the Open Policy Agent (OPA) project [Sty19]. OPA is a general-purpose policy engine and it can be used for authorization, admission control, and also for data filtering. In particular, OPA allows reasoning about data represented in structured JSON documents, and it is then suitable for evaluating NBI messages. OPA policies can be expressed using the *Rego* language [Sty19], which is a declarative language based on Datalog [CGT89].

We execute the OPA as a standalone server in a Linux container running inside IOx. In such a configuration, it can be queried for policy evaluation using HTTPS requests. In particular, the OPA server executes the following tasks:

1. It receives an NBI Message from the NBI.
2. It loads the Rego policies and context variables from the local repositories.
3. It creates scopes for the received NBI Message, updates the values of context variables, and evaluates the authorization policies against the NBI message.



4. It updates the context repository and returns an authorization decision to the NBI PEP. If required, it writes a log entry.

```

1. default allow = false
2. scopes[scope] { # Rule 1.
3.   scope := "IRRIGSYS"
4.   plc := ["PLC1", "IRRIGATOR"]
5.   plc[_] == input.message.context.deviceId}
6. context[ctx] { # Rule 2.
7.   input.topic == "Moisture"
8.   scopes[_] == "IRRIGSYS"
9.   input.message.context.dataSchemaId == "MoistureSchema"
10.  ts := to_number(input.message.context.timestamp)
11.  ctx := ctx_value("moisture", {"ts": ts, "value": input.message.
    ↳ message[0].values[0] })}
12. allow = true { # Rule 3.
13.  context
14.  input.topic == "Moisture"
15.  input.senderId[_] == "Modbus TCP"}
16. allow = true { # Rule 4.
17.  context
18.  input.topic == "Moisture"
19.  input.action == "subscribe"
20.  input.senderId[_] == "Irrigation App"}
21. allow = true { # Rule 5.
22.  input.topic == "Moisture"
23.  scopes[_] == "IRRIGSYS"
24.  input.action == "subscribe"
25.  input.senderId[_] == "Irrigation App"
26.  context["moisture"]["value"] < 30}

```

Figure 4.13: An example of Rego policy.

**Example 16.** Consider the rules introduced in Examples 12, 13, and 14. Figure 4.13 represents a translation of the above rules in the Rego language. Briefly, line 1 declares the default policy, i.e., a default deny configuration. Lines 2-5 represent a scope policy rule. It is expressed as a partial rule [Sty19], i.e., a statement that generates a value in the rule head (line 2) to be assigned to the `scopes` variable. This variable contains the set of message scopes. The rule body (line 4-5) checks for values of message properties according to constraints described in Examples 12. Lines 6-11 represent a partial rule used for context policy rules. In particular, `ctx_value` (line

11) is an internal function that updates the `context` variable with the `moisture` value if and only if the timestamp of the message is more recent than the saved one. Lines 14-26 represent authorization policy rules. Their heads (lines 12, 16, and 21) override the default deny value of the `allow` variable. The bodies (lines 15-16, 19-21, and 24-28) check the constraints described in Example 14. In particular, line 28 checks whether the actual value of the moisture, saved in the `context` variable, is less than the fixed value.  $\square$

```

1. {"result": [{
2.   "expressions": [{
3.     "value": {
4.       "allow": true,
5.       "context": [{
6.         "moisture": {
7.           "ts": 1579171129,
8.           "value": 90}}}],
9.     "scopes": [ "IRRIGSYS" ]}}]}]}

```

Figure 4.14: An example of OPA answer.

**Example 17.** Consider the legitimate `Modbus TCP` application reading the moisture value from the soil sensor and publishing it to the `Moisture` topic. Figure 4.14 depicts the answer from the OPA server. The message is authorized (line 4) since it matches rule 3 (line 12 of Figure 4.13). The answer also contains the updated `context` (line 5) and the `scopes` (line 9) related to the message.  $\square$

### 4.4.3 Experimental Evaluation

We use SDL to design the scenario described in Section 4.2.2. The specification substantially relies on the elements used in Section 3.5. The main difference is that we introduce the new types (i.e., `Software.PLC.ModBusTCP.Linux.Moisture`, and `Software.PLC.ModBusTCP.Linux.Irrigator`) for simulating PLCs. We implement the above components using Python and the `pyModbus` [Rip20] library. The entire SDL specification and the source code of the new components are available on GitHub [Rus20a]. Finally, we deploy a running instance of the testbed using CRACK.

We carried out two sets of experiments by executing *evilapp* on the irrigation system in place, in order to empirically assess both the reliability and the performance of DIOXIN.

In the first set of experiments, we executed *evilapp* on a standard IOx deployment, i.e., using only the native NBI implementation provided by Cisco.

Table 4.1: NBI Response Time in milliseconds.

	Default NBI Response Time	DIOXIN		
		PDP Response Time	NBI Response Time denied	NBI Response Time allowed
T1	448	47	339	460
T3/T4	409	46	286	433

In the second set of experiments we run evilapp after activating DIOXIN. Moreover, we introduced 2 scope, 1 context, and 9 authorization rules according to a default-deny policy. The scope rules label messages sent to the field devices and the remote cloud. The context rule save the actual value of the moisture level. The authorization rules define *i*) the admitted topic, *ii*) applications that can subscribe and publish on these topics, *iii*) the message format, i.e., valid data types and values, and *iv*) the soundness of messages concerning the saved moisture level and the freshness of their timestamp. The whole set of rules amounts to 86 Rego lines of code.

In both sets of experiments we used the malicious application executing 1000 repetitions of T2 attacks, i.e., subscribing to existing topics and tapping published messages, and 1000 repetitions of T3/T4 attacks, i.e., publishing messages to existing topics (see Section 4.3.2).

In the first set, we observed that the NBI accepted all legitimate and malicious messages. Moreover, we extracted the NBI response time for each message from the log of all requests.

In the second set, the experiments confirmed that the activation of DIOXIN denied all the malicious messages and allowed only legitimate requests. Table 4.1 shows the 90th-percentile of the PDP response time and the NBI response time for denied and allowed messages during the two repetitions of attacks.

Analyzing the PDP response time, we can observe that it represents on average 10% of the NBI response time. This implies that introducing DIOXIN generates a low overhead, even if NBI receives only allowed messages.

Below we answer the research question of Section 1.2.3.

**Answer to RQ6** We believe that Cyber Range can support the security assessment of critical assets. As a matter of fact, we created a realistic scenario for Fog Computing extending the IT infrastructure introduced in Section 3.1 and hosted by a Cyber Range. CRACK supported us in updating the infrastructure by reusing most of the existing components and adding the emulated PLCs. The resulting scenario has become a testbed for Fog Computing, merely connecting the physical Fog device under testing to a virtual network provided by the Cyber Range. This setup

allowed us to perform an extensive and intrusive security assessment without impacting production environments. Moreover, we used the Cyber Range for testing the solution we propose to remediate the found weaknesses.

# Chapter 5

## Related Work

In this chapter, we firstly present the related work concerning the ZenHackAdemy experience (see Section 5.1). In Section 5.2, we compare our proposal for Next Generation Cyber Ranges against other frameworks presented in the literature. Finally, in Section 5.3, we detail the related work about other solutions for policy enforcement in Fog Computing.

### 5.1 Hands-on and CTFs

The USENIX Workshops on Advances in Security Education constitutes an important venue to share educational experiences in the field of cybersecurity. Indeed, the workshops' papers introduce several case-studies on educational activities similar to ours. In many cases, gamification methodologies and techniques were selected to present cybersecurity scenarios, asking students to find possible solutions.

[CdR16] discusses an 11-week course addressing IoT security. Like in our case, each week presents a single topic, such as network protocols, web security, reverse engineering. During the course, students discovered a large number of vulnerabilities hidden inside the devices under analysis, and they learned how to carry on penetration testing activities on a set of unknown devices and programs. Another paper [CHRT17], of the same research group, proposes an experiment based on gamification. During an 11-week cybersecurity course, students played the role of newly hired IT security employees in charge of different tasks, presented as CTF-like exercises. Each exercise offers the chance to choose different options for advancing into the plot of the game. Depending on what the students decide, the plot evolves, and changes accordingly. Authors state that those students who actively followed the narration offered by the game scored better, as opposed to those who ignored the suggestions.

The goal of [VB16] is to understand the impact of the hints and the solutions given to the stu-

Table 5.1: Comparison with the related work.

	E	M	V	T	C	S
ALPACA [ECGB19]	●	●	●	○	●	○
EDURange [WBS <sup>+</sup> 15]	●	●	○	○	●	●
KYPO [VVO <sup>+</sup> 17]	●	●	○	○	●	●
CyRIS [BPT <sup>+</sup> 18]	●	●	○	○	○	●
ADLES [CdLGHK18]	●	●	●	○	●	●
SecGen [SSSAK <sup>+</sup> 17]	●	●	○	○	●	●
EZSetup [LNX17]	●	●	○	○	●	●
Tele-lab [WM12]	●	●	○	○	○	○
Labtainers [TI18]	●	○	○	○	●	○
DETER [PR10]	●	●	○	○	○	●
CRACK	●	●	●	●	●	●

dents who approach cybersecurity challenges. Actions performed by the students on the training platform were logged, producing data that were later analyzed. Results show that there was no evident correlation between the success rate of a challenge and the hints provided.

Flushman et al. [FGP15] set up a 10-week course, split into different modules. Each module covers a different cybersecurity topic. Students are organized in groups of four, mimicking a regular CTF team, and they play an Alternate Reality Game. Each exercise is provided with a fictional situation, blurring the boundary between the challenge and the inspired real-life scenario. At the end of each challenge, students are asked to reflect on their individual experience. These data have been used by the organizers to monitor the behavior of the participants and to discover problems in the hosting platform. Like in our experience, results show how gamification improved students' performances and awareness of computer security.

## 5.2 Cyber Ranges

The growing demand for cyber security professionals with hands-on skills is boosting the development of Cyber Ranges as well as training environments in general. In [YKG20] present a survey of Cyber Ranges and security testbeds. There, they also provide a taxonomy and an architectural model of a generic Cyber Range. CRACK (see Section 3.2.1) complies with their architectural requirements. Moreover, they mention a number of facilities, e.g., random traffic generators, that, although not yet implemented in CRACK, are compatible with our proposal.

In terms of exercises executed on a Cyber Range, Locked Shields [CCD19] is possibly the most

famous initiative. It is an annual event relying on a large, complex, and heterogeneous scenario. Locked Shields [CCD19] is probably the most famous live-fire exercise based on a Cyber Range. It is an annual event that leverages a large, complex, and heterogeneous scenario. The design phase of this exercise is based on a theater that is updated every year. While the execution phase lasts for a few days only, both design and testing require the effort of many experts for several months. Locked Shields is possibly the archetype of the Cyber Ranges relying on manual scenario development. As such, it matches the application conditions of CRACK, and thus, it is not an alternative to our approach.

Below, we compare CRACK against other frameworks presented in the literature in terms of the main features of CRACK, which we recall below.

- *Extensible* (E). The framework supports the seamless integration of new elements (e.g., vulnerabilities, software, and hardware).
- *Modular* (M). The elements defined within the framework can be composed without customizing them.
- *Verifiable* (V). The framework includes an engine for formally verifying the scenarios, e.g., the reachability of a target.
- *Testable* (T). The framework includes an engine for testing the scenarios, e.g., the exploitability of a vulnerability.
- *Compatible* (C). The framework relies on standard/well-established infrastructure development methodologies that are widely supported, e.g., TOSCA and Docker.
- *Scalable* (S). The framework allows for the definition of scenarios of different sizes, e.g., in terms of number of machines, and it scales with them.

We summarize the comparison results in Table 5.1. For each framework, we use ● and ○ to denote whether the feature is present or not, respectively. Moreover, we use ◐ to indicate that the feature is only partially present.

*ALPACA* [ECGB19] creates training scenarios involving multiple vulnerabilities. In particular, it uses a Prolog-based engine to generate a single virtual machine containing a set of vulnerabilities selected from a database. Vulnerabilities are organized according to a dependency lattice to ensure that an actual exploit exists. Unlike CRACK, ALPACA does not include a testing phase. Thus, when a new vulnerability is defined, to preserve the validity of the new scenarios, the entire lattice must be updated. It relies on a custom specification language, which is compiled to generate Ansible and Packer scripts.

*EDURange* [WBS<sup>+</sup>15] is a cloud-based framework for hosting cybersecurity scenarios. It also relies on a YAML-based specification language (inspired by the scenario description language

of [Fit14]) for designing cyber exercises. Analogously to our SDL, their language includes primitive types for modeling, e.g., software, artifacts, constraints, objectives, and actors. However, unlike our SDL, their language neither includes the notion of vulnerability nor supports the verification and testing of the scenarios. Finally, EDURange uses Terraform for the deployment phase.

The *KYPO* Cyber Range [VOC<sup>+</sup>17] uses structured JSON files for specifying goals, network topology, software, and scenario workflows. Each specification is translated to Ansible and Puppet scripts. Moreover, KYPO includes several predefined templates covering different categories of cybersecurity scenarios, e.g., Distributed Denial of Service (DDos) and phishing attacks. Verification and testing of the scenarios are not supported.

*CyRIS* [BPT<sup>+</sup>18] is a tool for the generation and management of cyber exercises. It also uses a YAML-based language for specifying users, firewall rules, and software running on each machine. CyRIS uses its own, customized engine for creating the virtual infrastructure. Finally, scenarios are neither verified nor tested.

*ADLES* [CdLGHK18] is a framework for reducing the effort of building, modifying, and deploying the virtual environments for cyber exercises. Also ADLES relies on a YAML-based specification language, but it compiles blueprints by means of a customized engine. Moreover, in the design stage, the authors claim a specification check phase performing several semantic checks. However, the details provided do not allow to outline if the framework is fully verifiable. Instead, testing of scenarios is not supported.

*SecGen* [SSSAK<sup>+</sup>17] creates vulnerable virtual machines to be used for learning penetration testing techniques. It includes a catalog of vulnerabilities that can be randomly selected based on constraints within the scenario definition. It uses an XML-based configuration language and leverages Puppet and Vagrant to provide the required virtual machines. Verification and testing of the scenarios are not supported by SecGen.

*EZSetup* [LNX17] is a web-based tool for creating and managing virtual environments for cybersecurity exercises. It relies on a YAML-based specification language for defining the network layout and connected virtual machines. By relying on an internal engine, EZSetup deploys the specified infrastructure on some cloud provider (among a set of supported ones). Then, an Ansible script is manually written to customize each virtual machine. Instead, verification and testing of exercises are not supported.

*Tele-lab* [WM12] is a platform for cyber security training using virtual labs. A virtual lab is a single virtual machine where the exercise is dynamically deployed. In particular, the designer defines a program, called agent, that manages the deployment by applying some parameters contained in an XML specification. Virtual labs and exercises are neither verified nor tested.

*Labtainers* [TI18] is a framework for creating and deploying Docker containers that host cybersecurity exercises. Each exercise is designed through an ad hoc scripting language. The language



allows specifying an *indicator*, i.e., a goal condition that is automatically checked at runtime. Labtainers exercises are executed on a single machine and they are neither verified not tested.

*DETER* [PR10] is a large scale, physical infrastructure consisting of a group of federated laboratories. The infrastructure is meant to execute security testbeds. *DETER* is managed through a YAML-based configuration language that has been used for defining complex cybersecurity training session. Again, no verification and testing operations are carried out by the framework.

Apart from the proposal described above, some other authors propose frameworks for helping the organization of Attack/Defense (A/D) Capture-The-Flag (CTF) competitions. For instance, SWPAG [TDG<sup>+</sup>17b] is a tool that provides a web interface for rapidly deploying an infrastructure for hosting A/D CTFs in the cloud. [RAPA16] works similarly, but it leverages application containers.

This kind of CTFs resembles the exercises played in a Cyber Range and their organization shares some of the issues discussed in this work. However, A/D CTFs are not executed on an arbitrarily complex infrastructure. Rather, they run on a dedicated platform designed to allow the participating teams to fairly compete by reciprocally attacking. For instance, a typical A/D CTF configuration consists of the same, vulnerable machine shared by all the teams. Each team looks for vulnerabilities that they can patch on their own machine and exploit on the others. Since this is a particular scenario, CRACK can be used to model and deploy it. Instead, CTF frameworks cannot be trivially extended to general purpose scenarios.

### 5.3 Fog Security

Since Fog Computing is an extension of the Cloud paradigm towards the edge, it inherits a combination of both Cloud and network security and privacy challenges (see, e.g., [RLM18, AAHC17, MM11]). Among them, the security of the interaction among Fog applications has driven the attention of the scientific community, mostly from an *access control* point of view [ZLY<sup>+</sup>18]. For example, [SMG15] theorized a distributed RMAC scheme for Fog Computing. In this scheme, security policies and attributes are preserved in a distributed policy information point (PIP) placed in the Cloud and the policy decision points (PDPs), that are in charge of making decision according to the access control policy, are implemented on Fog devices; the policy enforcement point (PEP), that enforces the access decisions, are instead implemented on the edge of the network.

Furthermore, [FWW<sup>+</sup>17] point out that Ciphertext-policy attribute-based encryption (ABE) can help to achieve data access control in fog-cloud systems. Hence, the authors propose an access control scheme based on a verifiable outsourced multi-authority.

However, one of the most relevant examples is the policy-based resource access control framework proposed by [DAT14], that adopts the eXtensible Access Control Markup Language (XACML)

to define formalized and refined operational, security and network policy specifications. However, the proposal is just a preliminary framework which does not include a lot of details regarding how to i) build the policy repository, ii) identify a user, iii) take a decision, and iii) protect the identity and grant data privacy. Furthermore, this work requires the inclusion of additional dedicated resources within Fog nodes that may introduce some operational latency and are weak against DoS attacks [KPQ17].

Similarly, [NPT18] propose a Model-Driven Security policy enforcement framework, named MDSIoT, for IoT tenant apps deployed at the Edge layer. The framework allows generating security enforcement code, called gatekeepers, for different kinds of IoT tenant apps, and deploying a tenant app with its corresponding gatekeeper, that acts as a local PDP-PEP, in the edge server. Although promising, MDSIoT requires substantial architectural changes, e.g., the introduction of an intermediate layer to intercept requests using the gatekeeper patterns, and it is mainly focused on architectural designs, policy modeling, and engineering approach without a definition of the run-time monitoring solution.

To the best of our knowledge, this work is the first methodology for the run-time enforcement of user-defined security policies on actual Fog applications.

# Chapter 6

## Conclusions

In this thesis, we deal with the design and implementation of Next Generation Cyber Ranges. We started introducing the Cyber Security Skills Shortage and the need to reduce this gap through training sessions that cover hands-on activities. For this reason, we considered the CTF competitions. To prove their effectiveness, we proposed to computer science and computer engineering students of our department a training activity, outside formal classes, involving CTF challenges. After two years of experience, we observed that this type of practical training attracts students. Moreover, we can claim that these activities raised their awareness of different aspects of Computer Security.

A Cyber Range can host CTF challenges and execute more complex and even more practical exercises to support training in large organizations. In particular, the overall quality of such exercises relies on a good design of the training scenario. To this aim, we presented CRACK, an open-source tool for modeling, verifying, and testing the scenarios for a Cyber Range. CRACK relies on the Scenario Definition Language (SDL), an extension of TOSCA that introduces several, new features. For instance, we defined novel types for modeling, e.g., vulnerabilities and goals, and a Datalog semantics. The Datalog translation enables the verification of the scenario and generates proof traces that drive the automatic testing process.

A further application of Cyber Ranges is creating testbeds for evaluating architecture and testing new security products in a controlled environment. In this work, we use a Cyber Range for executing a security assessment of Cisco IOx, a mainstream operating system for the emerging Fog Computing paradigm. We showed the inadequacy of current security mechanisms in IOx for actual Fog deployments. Then, we enhanced the native security mechanisms to support the runtime enforcement of user-defined security policies. Finally, we tested our proposal using the Cyber Range again. In this way, we were able to verify the effectiveness of our solution and the reduction of the attack surface for a IOx related deployment.

## **Acknowledgements**

This research was partly supported by the “Boeing-UNIGE Scholarship Project”, the Horizon 2020 project “Strategic Programs for Advanced Research and Technology in Europe” (SPARTA), and by the Italian Ministry of Defense PNRM project “UNAVOX”.

# Bibliography

- [AAHC17] Arwa Alrawais, Abdulrahman Alhothaily, Chunqiang Hu, and Xiuzhen Cheng. Fog computing for the internet of things: Security and privacy issues. *IEEE Internet Computing*, 21(2):34–42, 2017.
- [ABDN<sup>+</sup>17] Matej Artac, Tadej Borovssak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. DevOps: introducing infrastructure-as-code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 497–498. IEEE, 2017.
- [Apa20] Apache Software Foundation. HTTP Server Project. <http://httpd.apache.org/>, 2020.
- [ARH17] Syaiful Andy, Budi Rahardjo, and Bagus Hanindhito. Attack scenarios and security analysis of MQTT communication protocol in IoT system. In *2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, page 1–6, Sep 2017.
- [ARI19] ARIA TOSCA. Apache ARIA TOSCA orchestration engine. <http://ariatosca.incubator.apache.org/>, 2019. (Accessed on October 2019).
- [Ato19] Atos. Digital transformation with Atos alien4cloud and Cloudify. <https://atos.net/wp-content/uploads/2017/07/Alien4Cloud-and-Cloudify-White-paper.pdf>, 2019. (Accessed on October 2019).
- [BBH<sup>+</sup>13] Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. OpenTOSCA - A Runtime for TOSCA-Based Cloud Applications. In *Proceedings of the 11th International Conference on Service-Oriented Computing - Volume 8274, ICSOC 2013*, pages 692–695, Berlin, Heidelberg, 2013. Springer-Verlag.

- [BDF<sup>+</sup>18] Chiara Bodei, Pierpaolo Degano, Riccardo Focardi, Letterio Galletta, and Mauro Tempesta. Transcompiling firewalls. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust*, pages 303–324. Springer International Publishing, 2018.
- [Bel14] Thomas E Bell. Final Technical Report. Project Boeing SGS. Technical report, The Boeing Company, Seattle, WA (United States), 2014.
- [Ber14] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1:81–84, 09 2014.
- [BHL18] Kevin Bock, George Hughey, and Dave Levin. King of the hill: A novel cybersecurity competition for teaching penetration testing. In *2018 USENIX Workshop on Advances in Security Education (ASE 18)*, Baltimore, MD, 2018. USENIX Association.
- [BKEI08] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml 1.2 specification, 2008.
- [BPT<sup>+</sup>18] Razvan Beuran, Cuong Pham, Dat Tang, Ken-ichi Chinen, Yasuo Tan, and Yoichi Shinoda. Cybersecurity education and training support system: CyRIS. *IEICE Transactions on Information and Systems*, 101(3):740–749, 2018.
- [Bra17] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017.
- [BS14] Antonio Brogi and Jacopo Soldani. Reusing cloud-based services with TOSCA. In Erhard Plödereder, Lars Grunske, Eric Schneider, and Dominik Ull, editors, *44. Jahrestagung der Gesellschaft für Informatik, INFORMATIK 2014*, volume 232 of *LNI*, pages 235–246. GI, 2014.
- [BTS17] Antonio Brogi, Antonio Di Tommaso, and Jacopo Soldani. Sommelier: A Tool for Validating TOSCA Application Topologies. In *Model-Driven Engineering and Software Development - 5th International Conference, MODELSWARD 2017, Porto, Portugal, February 19-21, 2017, Revised Selected Papers*, pages 1–22, 2017.
- [Bye17] Charles C. Byers. Architectural imperatives for fog computing: Use cases, requirements, and architectural techniques for fog-enabled iot networks. *IEEE Communications Magazine*, 55:14–20, 2017.
- [Cam00] Jamie Cameron. Webmin a web-based system administration tool for unix. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '00*, page 33, USA, 2000. USENIX Association.

- [Can19] Canonical. Ubuntu Community Help Wiki: RootSudo. <https://help.ubuntu.com/community/RootSudo>, 2019. (Accessed on January 2020).
- [Car16] Pierre Carbonnelle. pyDatalog. <https://sites.google.com/site/pydatalog/>, 2016.
- [CBB14] Peter Chapman, Jonathan Burket, and David Brumley. Picocftf: A game-based computer security competition for high school students. In *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*, San Diego, CA, 2014. USENIX Association.
- [CCD19] CCDCOE. Locked Shields. <https://ccdcoe.org/exercises/locked-shields/>, 2019. (Accessed on October 2019).
- [CdLGHK18] Daniel Conte de Leon, Christopher E. Goes, Michael A. Haney, and Axel W. Krings. ADLES: Specifying, deploying, and sharing hands-on cyber-exercises. *Computers & Security*, 74(C):12–40, May 2018.
- [CdR16] Tom Chothia and Joeri de Ruiter. Learning From Others’ Mistakes: Penetration Testing IoT Devices in the Classroom. In *2016 USENIX Workshop on Advances in Security Education (ASE 16)*, Austin, TX, 2016. USENIX Association.
- [CF18] Pietro Colombo and Elena Ferrari. Access control enforcement within mqtt-based internet of things ecosystems. In *Proceedings of the 23rd ACM on Symposium on Access Control Models and Technologies*, pages 223–234, 2018.
- [CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, March 1989.
- [CGT90] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Syntax and Semantics of Datalog*, pages 77–93. Springer Berlin Heidelberg, Berlin, Heidelberg, 1990.
- [Che20a] Checkpoint. Cyber security report. <https://research.checkpoint.com/2020/the-2020-cyber-security-report/>, January 2020. (Accessed on October 2020).
- [Che20b] Chef Software. Chef. <https://www.chef.io/>, 2020. (Accessed on January 2020).
- [CHRT17] Tom Chothia, Sam Holdcroft, Andreea-Ina Radu, and Richard J Thomas. Jail, Hero or Drug Lord? Turning a Cyber Security Course Into an 11 Week Choose Your Own Adventure Story. In *2017 USENIX Workshop on Advances in Security Education (ASE 17)*, Vancouver, BC, 2017. USENIX Association.
- [Chu20] Kevin Chung. CTFd. <https://ctfd.io/>, 2020. (Accessed on October 2020).

- [Cis18] Cisco Systems Inc. Cisco IOx Data Sheet. <https://www.cisco.com/c/en/us/products/collateral/cloud-systems-management/iox/datasheet-c78-736767.html>, 2018. (Accessed on October 2020).
- [Cis19a] Cisco Systems Inc. Cisco IOx documentation. <https://developer.cisco.com/docs/iox/>, 2019. (Accessed on December 2019).
- [Cis19b] Cisco Systems Inc. Cisco IR829 Industrial Integrated Services Routers Data Sheet. <https://www.cisco.com/c/en/us/products/collateral/routers/829-industrial-router/datasheet-c78-734981.html>, 2019. (Accessed on December 2019).
- [Cis19c] Cisco Systems Inc. Networking Software (IOS & NX-OS). <https://www.cisco.com/c/en/us/products/ios-nx-os-software/index.html>, 2019. (Accessed on December 2019).
- [Clo19] Cloudify. Cloudify — Cloud and NFV Orchestration Based on TOSCA. <https://cloudify.co/>, 2019. (Accessed on October 2019).
- [CLU20] CLUSIT. Rapporto CLUSIT. <https://clusit.it/rapporto-clusit/>, April 2020. (Accessed on October 2020).
- [CPVV18] Marco Calabretta, Riccardo Pecori, Massimo Vecchio, and Luca Veltri. Mqtt-auth: A token-based solution to endow mqtt with authentication and authorization capabilities. 2018.
- [CTF20] CTFtime team. CTFtime. <https://ctftime.org/>, 2020. (Accessed on October 2020).
- [Cyb19] Cyberbit. Cyber Range Training and Simulation. <https://www.cyberbit.com/solutions/cyber-range/>, 2019. (Accessed on October 2019).
- [DAT14] Clinton Dsouza, Gail-Joon Ahn, and Marthony Taguinod. Policy-driven security management for fog computing: Preliminary framework and a case study. In *Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014)*, pages 16–23. IEEE, 2014.
- [DBZ70] Edward De Bono and Efrem Zimbalist. *Lateral thinking*. Penguin London, 1970.
- [DEC<sup>+</sup>11] Adam Doupé, Manuel Egele, Benjamin Caillat, Gianluca Stringhini, Gorkem Yakin, Ali Zand, Ludovico Cavedon, and Giovanni Vigna. Hit 'em where it hurts: A live security exercise on cyber situational awareness. In *ACM International Conference Proceeding Series*, pages 51–61, 2011.



- [DEGV01] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, September 2001.
- [Dew18] Robert S Dewar. Cybersecurity and cyberdefense exercises. Technical report, ETH Zurich, 2018.
- [DLR<sup>+</sup>19] Luca Demetrio, Giovanni Lagorio, Marina Ribaud, Enrico Russo, and Andrea Valenza. ZenHackAdemy: Ethical Hacking@DIBRIS. In *CSEDU (1)*, pages 405–413, 2019.
- [ECGB19] Joshua Eckroth, Kim Chen, Heyley Gatewood, and Brandon Belna. Alpaca: Building Dynamic Cyber Ranges with Procedurally-Generated Vulnerability Lattices. In *Proceedings of the 2019 ACM Southeast Conference*, ACM SE ’19, page 78–85, New York, NY, USA, 2019. Association for Computing Machinery.
- [Ecl19] Eclipse Foundation. Eclipse Mosquitto. <https://mosquitto.org/>, 2019. (Accessed on December 2019).
- [ENI16a] NCSS ENISA. Cybersecurity Skills Development in the EU. Technical report, 2016.
- [ENI16b] NCSS ENISA. Good practice guide designing and implementing national cyber security strategies. Technical report, 2016.
- [FGP15] Tanya Flushman, Mark Gondree, and Zachary N. J. Peterson. This is Not a Game: Early Observations on Using Alternate Reality Games for Teaching Security Concepts to First-Year Undergraduates. In *8th Workshop on Cyber Security Experimentation and Test (CSET 15)*, Washington, D.C., 2015. USENIX Association.
- [Fir20] Fireeye. M-Trends Reports. <https://www.fireeye.com/current-threats/annual-threat-report/mtrends.html>, February 2020. (Accessed on October 2020).
- [Fit14] Bryan K Fite. Simulating Cyber Operations: A Cyber Security Training Framework. *The SANS Institute*, 2014.
- [FSHB17] Vitaly Ford, Ambareen Siraj, Ada Haynes, and Eric Brown. Capture the flag unplugged: An offline cyber competition. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE ’17, pages 225–230, New York, NY, USA, 2017. ACM.
- [FWW<sup>+</sup>17] Kai Fan, Junxiong Wang, Xin Wang, Hui Li, and Yintang Yang. A secure and verifiable outsourced access control scheme in fog-cloud computing. *Sensors*, 17(7):1695, 2017.

- [GGV02] Georg Gottlob, Erich Grädel, and Helmut Veith. Datalog LITE: A Deductive Query Language with Linear Time Model Checking. *ACM Trans. Comput. Logic*, 3(1):42–79, January 2002.
- [Goo20] Google. Gruyere codelab. <https://google-gruyere.appspot.com/>, September 2020. (Accessed on September 2020).
- [Har12] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012.
- [Has20] HashiCorp. Terraform. <https://www.terraform.io/intro/>, 2020. (Accessed on January 2020).
- [HHM<sup>+</sup>17] S. T. Hamman, K. M. Hopkinson, R. L. Markham, A. M. Chaplik, and G. E. Metzler. Teaching game theory to improve adversarial thinking in cybersecurity students. *IEEE Transactions on Education*, 60(3):205–211, 2017.
- [Hiv19] HiveMQ. HiveMQ MQTT Broker. <https://www.hivemq.com/docs/4.2/hivemq/introduction.html>, 2019. (Accessed on December 2019).
- [HJN<sup>+</sup>11] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and JD Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58. ACM, 2011.
- [HK18] V Hauser and R Kessler. Thc-hydral penetration testing tools. <https://github.com/vanhauser-thc/thc-hydra>, 2018.
- [IDFF96] Roberto Ierusalimsky, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.
- [ISA20] ISACA. State of Cybersecurity. <https://www.isaca.org/go/state-of-cybersecurity-2020>, September 2020. (Accessed on October 2020).
- [J. 12] J. Richer, W. Mills, and H. Tschofenig. OAuth 2.0 message authentication code (MAC) tokens. <https://tools.ietf.org/html/draft-ietf-oauth-v2-http-mac-00>, 2012. (Accessed on December 2019).
- [JH12] M. Jones and D. Hardt. The OAuth 2.0 Authorization Framework: Bearer Token Usage. <https://tools.ietf.org/html/draft-ietf-oauth-v2-bearer-23>, 2012. (Accessed on December 2019).
- [JKAC18] Muhammad Aslam Jarwar, Muhammad Golam Kibria, Sajjad Ali, and Ilyoung Chong. Microservices in web objects enabled iot environment for enhancing reusability. In *Sensors*, 2018.

- [KLS12] Sven Köhler, Bertram Ludäscher, and Yannis Smaragdakis. Declarative Datalog Debugging for Mere Mortals. In Pablo Barceló and Reinhard Pichler, editors, *Datalog in Academia and Industry*, pages 111–122, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [KPQ17] Saad Khan, Simon Parkinson, and Yongrui Qin. Fog computing security: a review of current applications and security solutions. *Journal of Cloud Computing*, 6(1):19, Aug 2017.
- [KST<sup>+</sup>18] Carlos Kamienski, Juha-Pekka Soininen, Markus Taumberger, Stenio Fernandes, Attilio Toscano, Tullio Salmon Cinotti, Rodrigo Filev Maia, and Andre Torre Neto. Swamp: an iot-based smart water management platform for precision irrigation in agriculture. In *2018 Global Internet of Things Summit (GloTS)*, pages 1–6. IEEE, 2018.
- [LMMM<sup>+</sup>17] Antonio La Marra, Fabio Martinelli, Paolo Mori, Athanasios Rizos, and Andrea Saracino. Improving mqtt by inclusion of usage control. In *International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage*, pages 545–560. Springer, 2017.
- [LNX17] Yanyan Li, Dung Nguyen, and Mengjun Xie. Ezsetup: A novel tool for cybersecurity practices utilizing cloud resources. In *Proceedings of the 18th Annual Conference on Information Technology Education, SIGITE ’17*, page 53–58, New York, NY, USA, 2017. Association for Computing Machinery.
- [Lyo19] Gordon Lyon. Nmap scripting engine documentation. <https://nmap.org/nsedoc/scripts/http-userdir-enum.html>, 2019.
- [MF11] Alexey Melnikov and Ian Fette. The WebSocket Protocol. RFC 6455, December 2011.
- [MG11] Peter Mell and Timothy Grance. SP 800-145. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2011.
- [MH18] Daniel Miessler and J Haddix. Seclists. <https://github.com/danielmiessler/SecLists/tree/master/Passwords>, 2018.
- [Mic20] Microsoft. Microsoft Digital Defense Report. <https://www.microsoft.com/en-us/download/details.aspx?id=101738>, September 2020. (Accessed on October 2020).
- [MLM<sup>+</sup>07] B. E. Mullins, T. H. Lacey, R. F. Mills, J. E. Trechter, and S. D. Bass. How the Cyber Defense Exercise Shaped an Information-Assurance Curriculum. *IEEE Security Privacy*, 5(5):40–49, 2007.

- [MM11] M. Migliardi and A. Merlo. Modeling the energy consumption of distributed ids: A step towards green security. pages 1452–1457, 2011.
- [MNE<sup>+</sup>17] Avijit Mathur, Thomas Newe, Walid Elgenaidi, Muzaffar Rao, Gerard Dooly, and Daniel Toal. A secure end-to-end iot solution. *Sensors and Actuators A: Physical*, 263:291–299, 2017.
- [NIP<sup>+</sup>16] Aimaschana Niruntasukrat, Chavee Issariyapat, Panita Pongpaibool, Koonlachart Meesublak, Pramrudee Aiumsupucgul, and Anun Panya. Authorization mechanism for mqtt-based internet of things. In *2016 IEEE International Conference on Communications Workshops (ICC)*, pages 290–295. IEEE, 2016.
- [NIS18] NIST. Cyber Ranges. [https://www.nist.gov/system/files/documents/2018/02/13/cyber\\_ranges.pdf](https://www.nist.gov/system/files/documents/2018/02/13/cyber_ranges.pdf), 2018. (Accessed on January 2020).
- [NIS20] NIST. Glossary: vulnerability. <https://csrc.nist.gov/glossary/term/vulnerability>, 2020. (Accessed on January 2020).
- [NPT18] Phu H Nguyen, Phu H Phung, and Hong-Linh Truong. A security policy enforcement framework for controlling iot tenant applications in the edge. In *Proceedings of the 8th International Conference on the Internet of Things*, page 4. ACM, 2018.
- [NSS14] Dmitry Namiot and Manfred Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27, 2014.
- [OAS19] OASIS. OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC. <https://www.oasis-open.org/committees/tosca/>, 2019. (Accessed on October 2019).
- [OGA05] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. MulVAL: A Logic-based Network Security Analyzer. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM’05, page 8, USA, 2005. USENIX Association.
- [Ope19a] OpenStack. DevStack. <https://docs.openstack.org/devstack/latest/>, 2019.
- [Ope19b] OpenStack. Heat Translator. <https://wiki.openstack.org/wiki/Heat-Translator>, 2019. (Accessed on October 2019).
- [Pal20] Pallets. Werkzeug. <https://werkzeug.palletsprojects.com/en/0.15.x/debug/>, 2020. (Accessed on January 2020).

- [PPR<sup>+</sup>19] Andrea Palmieri, Paolo Prem, Silvio Ranise, Umberto Morelli, and Tahir Ahmad. MQTTSA: A Tool for Automatically Assisting the Secure Deployments of MQTT Brokers. In *2019 IEEE World Congress on Services (SERVICES)*, volume 2642–939X, page 47–53, Jul 2019.
- [PR10] Peter Peterson and Peter L Reiher. Security exercises for the online classroom with deter. In *CSET*, 2010.
- [Pup20] Puppet. puppet. <https://puppet.com/>, 2020. (Accessed on January 2020).
- [RAPA16] Arvind S. Raj, Bithin Alangot, Seshagiri Prabhu, and Krishnashree Achuthan. Scalable and lightweight CTF infrastructures using application containers (pre-recorded presentation). In *2016 USENIX Workshop on Advances in Security Education (ASE 16)*, Austin, TX, August 2016. USENIX Association.
- [RBL19] Matt Rutkowski, Luc Boutier, and Chris Lauwers. TOSCA Simple Profile in YAML Version 1.2. Technical report, OASIS, January 2019.
- [RCA18] Enrico Russo, Gabriele Costa, and Alessandro Armando. Scenario Design and Validation for Next Generation Cyber Ranges. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–4. IEEE, 2018.
- [RCA20] Enrico Russo, Gabriele Costa, and Alessandro Armando. Building Next Generation Cyber Ranges with CRACK. *Computers & Security*, 95:101837, 2020.
- [Red20] Red Hat. Ansible. <https://www.ansible.com/overview/>, 2020. (Accessed on January 2020).
- [RHP<sup>+</sup>16] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, and Piotr Mardziel. Build it, break it, fix it: Contesting secure development. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS ’16*, pages 690–703, New York, NY, USA, 2016. ACM.
- [Rip20] RiptideIO. PyModbus - A Python Modbus Stack. <https://github.com/riptideio/pymodbus>, 2020. (Accessed on October 2020).
- [RLM18] Rodrigo Roman, Javier Lopez, and Masahiro Mambo. Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges. *Future Generation Computer Systems*, 78:680–698, 2018.
- [Rus20a] Enrico Russo. CRACK. <https://github.com/enricorusso/CRACK>, October 2020.
- [Rus20b] Enrico Russo. Crack. <https://doi.org/10.5281/zenodo.4158113>, October 2020.

- [RVAM20] Enrico Russo, Luca Verderame, Alessandro Armando, and Alessio Merlo. DIOXIN: Runtime Security Policy Enforcement of Fog Applications. *International Journal of Grid and Utility Computing*, 2020. In press.
- [RVM19] E. Russo, L. Verderame, and A. Merlo. Towards policy-driven monitoring of fog applications. In *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 93–97, June 2019.
- [S<sup>+</sup>99] Andy Swales et al. Open modbus/tcp specification. *Schneider Electric*, 29, 1999.
- [SAN19] SANS. NetWars. <https://www.sans.org/netwars/>, 2019. (Accessed on October 2019).
- [SCC19] Wei-Tsung Su, Wei-Cheng Chen, and Chao-Chun Chen. An extensible and transparent thing-to-thing security enhancement for mqtt protocol in iot environment. In *2019 Global IoT Summit (GIoTS)*, pages 1–4. IEEE, 2019.
- [Sis19] Sistema di Informazione per la Sicurezza della Repubblica Italiana. Relazione sulla Politica dell’Informazione per la Sicurezza. <https://www.sicurezzanazionale.gov.it/sisr.nsf/wp-content/uploads/2020/03/RELAZIONE-ANNUALE-2019-4.pdf>, 2019. (Accessed on July 2020).
- [SMG15] Stavros Salonikias, Ioannis Mavridis, and Dimitris Gritzalis. Access control issues in utilizing fog computing for transport infrastructure. In *International Conference on Critical Information Infrastructures Security*, pages 15–26. Springer, 2015.
- [SO18] E. Seker and H. H. Ozbenli. The Concept of Cyber Defence Exercises (CDX): Planning, Execution, Evaluation. In *2018 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, pages 1–9, 2018.
- [SPB19] Abhishek Singh, Ashish Payal, and Sourabh Bharti. A walkthrough of the emerging iot paradigm: Visualizing inside functionalities, key features, and open issues. *Journal of Network and Computer Applications*, 143:111–151, 2019.
- [SSSAK<sup>+</sup>17] Z. Cliffe Schreuders, Thomas Shaw, Mohammad Shan-A-Khuda, Gajendra Ravichandran, Jason Keighley, and Mihai Ordean. Security Scenario Generator (SecGen): A Framework for Generating Randomly Vulnerable Rich-scenario VMs for Learning Computer Security and Hosting CTF Events. In *2017 USENIX Workshop on Advances in Security Education (ASE 17)*, Vancouver, BC, August 2017. USENIX Association.

- [Sty19] Styra Inc. Open Policy Agent documentation. <https://www.openpolicyagent.org/docs/v0.16.1/>, 2019. (Accessed on December 2019).
- [TDG<sup>+</sup>17a] Erik Trickel, Francesco Disperati, Eric Gustafson, Faezeh Kalantari, Mike Mabey, Naveen Tiwari, Yeganeh Safaei, Adam Doupé, and Giovanni Vigna. Shell we play a game? ctf-as-a-service for security education. In *2017 USENIX Workshop on Advances in Security Education (ASE 17)*, Vancouver, BC, 2017. USENIX Association.
- [TDG<sup>+</sup>17b] Erik Trickel, Francesco Disperati, Eric Gustafson, Faezeh Kalantari, Mike Mabey, Naveen Tiwari, Yeganeh Safaei, Adam Doupé, and Giovanni Vigna. Shell we play a game? ctf-as-a-service for security education. In *2017 USENIX Workshop on Advances in Security Education (ASE 17)*, Vancouver, BC, 2017. USENIX Association.
- [The19a] The MITRE Corporation. CAPEC-151: Identity Spoofing. <https://capec.mitre.org/data/definitions/151.html>, 2019. (Accessed on December 2019).
- [The19b] The MITRE Corporation. CAPEC-157: Sniffing Attacks. <https://capec.mitre.org/data/definitions/157.html>, 2019. (Accessed on December 2019).
- [The19c] The MITRE Corporation. CAPEC-94: Man in the Middle Attack. <https://capec.mitre.org/data/definitions/94.html>, 2019. (Accessed on December 2019).
- [The19d] The MITRE Corporation. CVE-2019-11043. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11043>, 2019. (Accessed on January 2020).
- [The19e] The MITRE Corporation. CVE-2019-15107. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-15107>, 2019. (Accessed on January 2020).
- [The20a] The Apache Software Foundation. HTTP Server Project - Reverse Proxy Guide. [https://httpd.apache.org/docs/2.4/howto/reverse\\_proxy.html](https://httpd.apache.org/docs/2.4/howto/reverse_proxy.html), 2020. (Accessed on January 2020).
- [The20b] The MITRE Corporation. CWE-250: Execution with Unnecessary Privileges. <https://cwe.mitre.org/data/definitions/250.html>, 2020. (Accessed on January 2020).

- [The20c] The MITRE Corporation. CWE-441: Unintended Proxy or Intermediary ('Confused Deputy'). <https://cwe.mitre.org/data/definitions/441.html>, 2020. (Accessed on January 2020).
- [The20d] The MITRE Corporation. T1087: Account Discovery. <https://attack.mitre.org/techniques/T1087/>, 2020. (Accessed on January 2020).
- [The20e] The MITRE Corporation. T1110: Brute Force . <https://attack.mitre.org/techniques/T1110/>, 2020. (Accessed on January 2020).
- [The20f] The MITRE Corporation. T1169: Sudo. <https://attack.mitre.org/techniques/T1169/>, 2020. (Accessed on January 2020).
- [The20g] The MITRE Corporation. TA0004: Privilege Escalation. <https://attack.mitre.org/tactics/TA0004/>, 2020. (Accessed on January 2020).
- [The20h] The PHP Group. PHP. <https://www.php.net/>, 2020.
- [the20i] the Wireshark Foundation. Wireshark. <https://www.wireshark.org/>, 2020. (Accessed on October 2020).
- [TI18] Michael F Thompson and Cynthia E Irvine. Individualizing cybersecurity lab exercises with labtainers. *IEEE Security & Privacy*, 16(2):91–95, 2018.
- [Tob92] RM Tobias. Defining non-formal and community education. 1992.
- [UK 19] UK Government. Cyber Security Breaches Survey 2019. [https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment\\_data/file/875799/Cyber\\_Security\\_Breaches\\_Survey\\_2019\\_-\\_Main\\_Report\\_-\\_revised.pdf](https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/875799/Cyber_Security_Breaches_Survey_2019_-_Main_Report_-_revised.pdf), 2019. (Accessed on July 2020).
- [VB16] Jan Vykopal and Miloš Barták. On the design of security games: From frustrating to engaging learning. In *2016 USENIX Workshop on Advances in Security Education (ASE 16)*, Austin, TX, 2016. USENIX Association.
- [VOC<sup>+</sup>17] Jan Vykopal., Radek Oslejsek., Pavel Celeda., Martin Vizvary., and Daniel Tovarnak. Kypo cyber range: Design and use cases. In *Proceedings of the 12th International Conference on Software Technologies - Volume 1: ICSOFT*., pages 310–321. INSTICC, SciTePress, 2017.
- [VVO<sup>+</sup>17] J. Vykopal, M. Vizvary, R. Oslejsek, P. Celeda, and D. Tovarnak. Lessons learned from complex hands-on defence exercises in a cyber range. In *2017 IEEE Frontiers in Education Conference (FIE)*, pages 1–8, Oct 2017.



- [WBS<sup>+</sup>15] Richard S. Weiss, Stefan Boesen, James F. Sullivan, Michael E. Locasto, Jens Mache, and Erik Nilsen. Teaching Cybersecurity Analysis Skills in the Cloud. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education, SIGCSE '15*, pages 332–337, New York, NY, USA, 2015. ACM.
- [WCC18] SeongIl Wi, Jaeseung Choi, and Sang Kil Cha. Git-based CTF: A simple and effective approach to organizing in-course attack-and-defense security competition. In *2018 USENIX Workshop on Advances in Security Education (ASE 18)*, Baltimore, MD, 2018. USENIX Association.
- [WM12] Christian Willems and Christoph Meinel. Online assessment for hands-on cyber security training in a virtual lab. In *Proceedings of the 2012 IEEE Global Engineering Education Conference (EDUCON)*, pages 1–10. IEEE, 2012.
- [Wor20] WordPress. WordPress. <https://wordpress.org/>, 2020.
- [YKG20] Muhammad Mudassar Yamin, Basel Katt, and Vasileios Gkioulos. Cyber ranges and security testbeds: Scenarios, functions, tools and architecture. *Computers & Security*, 88:101636, 2020.
- [YM13] F. Yang and S. Manoharan. A security analysis of the oauth protocol. In *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 271–276, 2013.
- [Zan19] Tommaso De Zan. MIND THE GAP: the Cyber Security Skills Shortage and Public Policy Interventions. Technical report, 2019.
- [ZLY<sup>+</sup>18] Peng Zhang, Joseph K Liu, F Richard Yu, Mehdi Sookhak, Man Ho Au, and Xiapu Luo. A survey on access control in fog computing. *IEEE Communications Magazine*, 56(2):144–149, 2018.